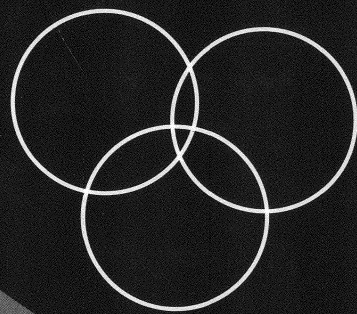


PROGRAMANDO COM

ASP.NET MVC



APRENDA A DESENVOLVER
APLICAÇÕES WEB UTILIZANDO
A ARQUITETURA MVC

novatec

Alfredo Lotar

Programando com ASP.NET MVC

**Aprenda a desenvolver aplicações web
utilizando a arquitetura MVC**

Programando com ASP.NET MVC

**Aprenda a desenvolver aplicações web
utilizando a arquitetura MVC**

Alfredo Lotar

Copyright© 2011 da Novatec Editora Ltda.

Todos os direitos reservados e protegidos pela Lei 9610 de 19/02/1998.

É proibida a reprodução desta obra, mesmo parcial, por qualquer processo, sem prévia autorização, por escrito, do autor e da Editora.

Editor: Rubens Prates

Editoração eletrônica: Camila Kuwabata e Carolina Kuwabata

Revisão gramatical: Débora Facin

Capa: Victor Bittow

ISBN: 978-85-7522-283-6

Histórico de impressões:

Fevereiro/2014	Terceira reimpressão
Janeiro/2013	Segunda reimpressão
Abril/2012	Primeira reimpressão
Outubro/2011	Primeira edição

Novatec Editora Ltda.

Rua Luís Antônio dos Santos 110

02460-000 – São Paulo, SP – Brasil

Tel.: +55 11 2959-6529

Fax: +55 11 2950-8869

Email: novatec@novatec.com.br

Site: www.novatec.com.br

Twitter: twitter.com/novateceditora

Facebook: facebook.com/novatec

LinkedIn: linkedin.com/in/novatec

Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

Lotar, Alfredo
Programando com ASP.NET MVC : aprenda a
desenvolver aplicações web utilizando a
arquitetura MVC / Alfredo Lotar. -- São Paulo :
Novatec Editora, 2011.

Bibliografia.
ISBN 978-85-7522-283-6

1. Active Server Pages 2. Microsoft.NET
Framework 3. Padrões de software 4. Web sites -
Desenvolvimento I. Título.

11-11236

CDD-006.7882

Índices para catálogo sistemático:

1. ASP.NET MVC : Tecnologia de desenvolvimento
de sites da Web : Computadores : Processamento
de dados 006.7882
OG20140211

Dedico este livro à minha família e amigos, e também a todos os leitores que entraram em contato comigo por e-mail, Facebook e Twitter, ansiosos por um livro de ASP.NET MVC.

Sumário

Capítulo 1 ■ Introdução ASP.NET MVC.....	15
1.1 Para quem é este livro?	16
1.2 Por que outro ASP.NET?	16
1.3 ASP.NET MVC é difícil?.....	16
1.4 Quando usar o ASP.NET Web Forms	17
1.5 Quando usar o ASP.NET MVC.....	17
1.6 Os dois ASP.NET funcionam juntos?	17
1.7 Recursos do ASP.NET MVC	18
1.8 Prepare seu computador	18
1.8.1 Preparando o IIS	18
1.8.2 Acessando uma aplicação ASP.NET MVC	19
1.8.3 Criando um diretório virtual	19
1.9 Desenvolvendo uma nova aplicação	20
1.10 Executando a aplicação	23
1.11 Controladores	24
1.11.1 Método com parâmetros.....	27
1.12 Master pages	28
1.12.1 Configurando a master page.....	30
1.13 Views	33
1.13.1 Exibindo informações num view.....	35
1.13.2 Acrescentando um link ao view	36
1.14 Models.....	38
1.14.1 Acrescentando uma nova classe	39
1.14.2 Criando uma instância da classe Cidades	41
1.14.3 Acrescentando um view	42
Capítulo 2 ■ Controladores	44
2.1 Desenvolvendo uma nova aplicação	44
2.2 Classe controladora	47
2.2.1 Parâmetros	48
2.2.2 Usando o atributo AcceptVerbs	51
2.2.3 Atributo ActionName	53

2.2.4 Atributo NonAction.....	54
2.2.5 Atributo Bind	54
2.2.6 Método HandleUnknownAction	55
2.2.7 Atributo HandleError	56
2.2.8 Criando atributos.....	57
2.2.9 Atributo SessionState.....	58
2.2.10 Tipos retornados de ActionResult.....	60
2.3 Cache	70
2.3.1 Parâmetro Duration	71
2.3.2 Cache com múltiplas versões de uma página.....	71
2.3.3 Cache parcial.....	76
2.3.4 Adicionando itens para o cache	76
2.3.5 Extraíndo itens do cache	77
2.3.6 Excluindo itens do cache.....	77
2.4 Controladores assíncronos	78
2.4.1 Atributos com métodos assíncronos.....	79
2.4.2 Outras operações assíncronas	80
Capítulo 3 ■ View.....	81
3.1 Desenvolvendo uma nova aplicação	81
3.2 Adicionando um view.....	83
3.3 View tipados.....	85
3.4 View predefinidos	86
3.4.1 Template List	86
3.4.2 Template Details	89
3.4.3 Template Create.....	91
3.4.4 Template Edit	94
3.4.5 Template Delete	97
3.4.6 Criando ou alterando templates	99
3.5 Partial view	100
3.5.1 Acrescentando um novo partial view	100
3.5.2 Exibindo o conteúdo de um partial view	102
3.5.3 Associando classes a um user control.....	104
Capítulo 4 ■ Model.....	107
4.1 Desenvolvendo uma nova aplicação	107
4.2 Usando ADO.NET	107
4.3 ADO.NET Entity Framework	111
4.3.1 Exibindo registros	115
4.3.2 Exibindo detalhes de um registro	116
4.3.3 Adicionando um novo registro	119

4.3.4 Editando registros.....	119
4.3.5 Excluindo registros	121
4.4 Camada de dados.....	122
4.4.1 Interface ICategoriesRepository	122
4.4.2 Classe CategoriesRepository	122
4.4.3 Invoque os métodos da interface	126
4.5 Repositório genérico.....	128
4.5.1 A interface IGenericRepository	129
4.5.2 Classe EntityFrameworkRepository.....	130
4.5.3 Invocando os métodos da interface generics.....	135

Capítulo 5 ■ Razor.....139

5.1 Variáveis	139
5.2 Comentários	141
5.3 Instruções if	141
5.4 Instrução switch	142
5.5 Instrução for.....	143
5.6 Instrução foreach	143
5.7 Propriedades e métodos	144
5.8 Manipular exceções	144
5.9 Conversões	145
5.10 Operadores	145
5.11 Href	146
5.12 Associando classes.....	146
5.13 Adicionando um layout	147
5.14 Criando um novo view Razor.....	149
5.15 Páginas com Visual consistente	151
5.15.1 Páginas de layout e o método RenderPage	153
5.15.2 Seções.....	154
5.16 PageData[key], PageData[index], Page.....	155
5.17 WebGrid Helper.....	155
5.17.1 Formatando colunas.....	158
5.17.2 Aplicando uma folha de estilo CSS	158
5.17.3 Outros parâmetros.....	160
5.17.4 Paginação de registros	161
5.17.5 Seleção de linhas	162
5.17.6 Extraíndo informações do WebGrid	164
5.17.7 WebGrid Helper e Ajax	165
5.18 WebImage Helper	165
5.19 Chart Helper.....	167
5.19.1 Usando outras fontes de dados	168
5.19.2 Exibindo gráficos em uma página	169

5.193 Personalizar a aparência	169
5.194 Salvando um gráfico	170
5.195 Colocando um gráfico no cache	171
5.20 WebMail Helper	172
5.20.1 Enviando um e-mail com anexos.....	173
5.20.2 Acrescentando cabeçalhos à mensagem.....	177
5.21 Crypto Helper	177

Capítulo 6 ■ Roteamento de URLs.....179

6.1 Modelo de URLs.....	179
6.2 Restrições.....	181
6.2.1 Método HttpMethodConstraint	181
6.2.2 Restrições personalizadas.....	182
6.3 Definindo URLs	182
6.3.1 Gerando links com ActionLink	183
6.3.2 Gerando links com RouteLink	184
6.3.3 Redirecionamentos	185
6.3.4 Obtendo uma URL.....	186
6.4 Áreas.....	187
6.4.1 Conflitos entre controladores.....	189
6.4.2 Link para diferentes áreas	190
6.4.3 Link para a raiz da aplicação.....	190

Capítulo 7 ■ HTML Helpers191

7.1 Desenvolvendo uma nova aplicação	191
7.2 Formulários	191
7.3 Preparando um formulário.....	197
7.4 Exibindo legendas.....	198
7.4.1 Método Label.....	198
7.4.2 Método LabelFor	199
7.4.3 Método LabelForModel	199
7.5 Caixas de texto	199
7.5.1 Método TextBox	200
7.5.2 Método TextBoxFor	201
7.5.3 Método TextArea	201
7.5.4 Método TextAreaFor	202
7.5.5 Métodos Hidden e HiddenFor.....	203
7.5.6 Método Password e PasswordFor.....	203
7.6 Caixas de seleção	203
7.7 Botões de rádio	205

7.8 Caixa de combinação.....	206
7.8.1 Método DropDownListFor	208
7.9 Caixa de listagem	209
7.9.1 Método ListBoxFor.....	211
7.10 Métodos de edição	211
7.10.1 Método EditorForModel	211
7.10.2 Método Editor.....	213
7.10.3 Método EditorFor.....	213
7.11 Exibindo Texto.....	213
7.12 Ativando e desativando elementos HTML.....	214
7.12.1 Método Encode	214
7.12.2 Método AttributeEncode	214
7.12.3 Método Raw	215
7.13 Formulário de inserção de dados.....	215
7.14 Formulário de atualização de dados	218
7.15 Criando HTML Helpers	222
7.15.1 HTML helpers inline.....	224
7.15.2 Classe TagBuilder	225
7.15.3 Classe HtmlTextWriter	228
 Capítulo 8 ■ Verificações e validações	230
8.1 Desenvolvendo uma nova aplicação	230
8.2 Atributos.....	231
8.2.1 Atributo Required.....	232
8.2.2 Atributo StringLength	233
8.2.3 Atributo Display.....	233
8.2.4 Atributo DataType.....	233
8.2.5 Atributo DisplayFormat	233
8.2.6 Atributo Range.....	234
8.2.7 Atributo RegularExpression	234
8.2.8 Atributo Compare	234
8.2.9 Atributo Remote.....	235
8.3 Validação no cliente	236
8.4 Exibindo mensagens de erro	237
8.4.1 Mensagens de erro com ValidationMessage	239
8.4.2 Mensagens de erro com ValidationMessageFor.....	240
8.4.3 Aplicando folhas de estilo CSS.....	241
8.5 Data Annotations Extensions.....	241
8.6 Atributos personalizados	244
8.6.1 Validação do lado do cliente	245
8.6.2 Aplicando o atributo personalizado.....	246

8.7 Validando propriedades do ADO.NET Entity Framework	247
8.8 Validação com ModelState	248
8.9 Internacionalização de mensagens de erro	249
8.9.1 Resource files – arquivos de recursos	250
8.9.2 Determine a cultura	251
8.9.3 Como aplicar mensagens de erro a uma propriedade	253
8.9.4 Testando a aplicação	254
Capítulo 9 – Filtros de ação	256
9.1 Filtros de autorização	256
9.2 Filtros de ação	257
9.3 Filtros de resultado	257
9.4 Filtros de exceções	257
9.5 Desenvolvendo uma nova aplicação	257
9.6 Criando um filtro de ação personalizado	258
Capítulo 10 – AJAX	262
10.1 Desenvolvendo uma nova aplicação	262
10.2 Ativando Unobtrusive Ajax	262
10.3 Método Ajax.ActionLink	263
10.4 Enviando um formulário de forma assíncrona	267
10.5 Mensagens de confirmação	269
10.6 Propriedade OnSuccess	270
10.7 Acrescentando elementos visuais	271
10.8 Método IsAjaxRequest	271
Capítulo 11 – Segurança de aplicações ASP.NET MVC	272
11.1 Criando uma nova aplicação	273
11.2 Criando o banco de dados de segurança	278
11.3 Alterando o arquivo de configuração	278
11.4 Gerenciando a aplicação	281
11.4.1 A aplicação	283
11.5 ADO.NET Entity Framework	287
11.5.1 Validando as propriedades de autenticação	289
11.6 Segurança de acesso ao código	291
11.7 Proteção extra	292
11.7.1 Ataques CSRF	292
11.7.2 Atributos ValidateInput e AllowHtml	294
11.8 Validando entradas	295
11.9 Autenticação	295

11.10 Autorização.....	296
11.11 Dados confidenciais.....	296
11.12 Manipulação de parâmetros	297
11.13 Gerenciamento de exceções.....	297
11.14 Log de erros	297
11.15 Aplicando SSL.....	298
11.15.1 Configurando SSL no Servidor web	298
11.15.2 Aplicando SSL a um diretório	301

Capítulo 12 ■ Desenvolvendo um website com o ASP.NET MVC303

12.1 Especificações	303
12.1.1 Quais recursos devo usar.....	305
12.2 Criando uma nova aplicação.....	306
12.3 Telas da aplicação.....	306
12.4 Layout	312
12.5 Desenvolvendo a camada de dados	322
12.6 Menu	326
12.6.1 Validação e formatação de dados	329
12.7 Exibindo os produtos.....	330
12.7.1 Validação e formatação de dados	334
12.8 Mecanismo de busca.....	335
12.9 Formulário de contato	339
12.9.1 O banco de dados.....	339
12.9.2 Projetando o formulário de contato	340
12.9.3 O método AddContato	342
12.9.4 Validação e formatação	344
12.10 Cadastro de usuários	345
12.10.1 Validando a entrada.....	349
12.10.2 Criando a tabela.....	350
12.10.3 ADO.NET Entity Framework	351
12.10.4 Validando as propriedades.....	351
12.10.5 Métodos de controlador	352
12.10.6 Projetando o formulário de cadastro	353
12.10.7 Informações de criptografia	356
12.11 Login da aplicação	359
12.11.1 Formulário de login.....	362
12.11.2 Enviando uma nova senha.....	367
12.12 Carrinho de compras.....	370
12.12.1 Criando as tabelas.....	370
12.12.2 ADO.NET Entity Framework	371
12.12.3 Validação e formatação.....	372
12.12.4 Classe controladora	373

12.12.5 Adicione outros recursos	381
12.12.6 Publicando a aplicação.....	382
12.12.7 Unidade de teste	383
Referências	388
Sites ASP.NET em português	388
Sites ASP.NET em inglês.....	388
Índice remissivo	389

Introdução ASP.NET MVC

O ASP.NET MVC fornece, por meio de design patterns, uma maneira poderosa e alternativa para criar websites ASP.NET dinâmicos.

O ASP.NET MVC implementa o pattern MVC e separa a aplicação em três componentes: model, controller e view.

- O model contém o código da camada de dados.
- O controller recebe as requisições do usuário.
- O view implementa o design da aplicação.

Observe a figura 1.1:

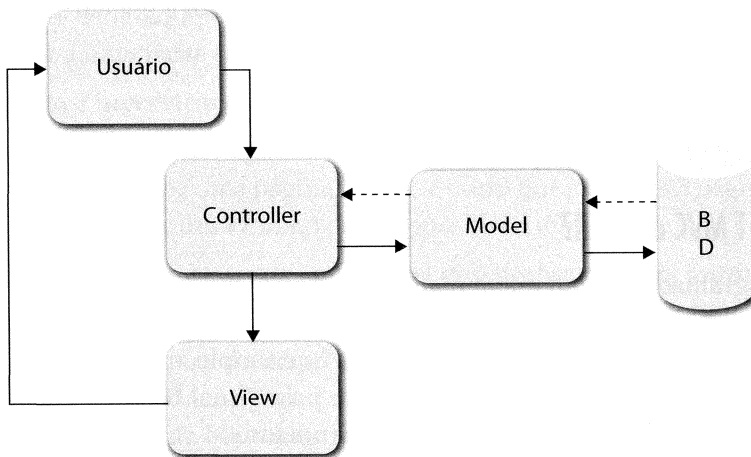


Figura 1.1 – Relação entre o usuário, o controller, o model e o view.

Na figura 1.1 vemos a relação entre o controller, o model e o view. O controller aceita a requisição do usuário, acessa o view e/ou o model. O model acessa o banco de dados. Ao final, o conteúdo do view é retornado ao usuário.

Obs.: no decorrer deste livro usamos a palavra controlador em vez de controller. As palavras model e view não foram traduzidas.

1.1 Para quem é este livro?

Bem, este não é um livro para iniciantes em programação, ou seja, é necessário conhecimento prévio de C# e/ou ASP.NET Web Forms.

Neste livro não abordamos a sintaxe C#, o ASP.NET Web Forms e as classes do .NET Framework, pois o foco do livro é especificamente o ASP.NET MVC.

Para testar os exemplos abordados no livro, baixe os arquivos de exemplo no website da Novatec: <http://www.novatec.com.br/downloads.php>

Obs.: ASP.NET Web Forms é a denominação usada para o ASP.NET tradicional, com base em formulários e controles. Este nome visa apenas a diferenciá-lo do ASP.NET MVC.

1.2 Por que outro ASP.NET?

Porque o ASP.NET Web Forms apresenta uma série de limitações e problemas, por exemplo:

- Gera páginas muito grandes, afetando o carregamento das páginas e o tráfego da rede.
- Temos pouco controle sobre o HTML gerado.
- Os Web Server Controls são processados no servidor e podem afetar o desempenho da aplicação.
- Dificuldade em realizar testes na aplicação.
- Não tem real separação entre o código e o design.

1.3 ASP.NET MVC é difícil?

Alguns programadores com quem mantenho contato pela Internet afirmam: o ASP.NET MVC é muito difícil comparado ao ASP.NET Web Forms. Na minha opinião o ASP.NET MVC não é difícil, mas diferente. Por exemplo, o ASP.NET Web Forms é semelhante ao modelo de programação usado pelo Visual Basic ou Delphi, ao qual a maioria dos programadores atuais está acostumado.

No ASP.NET MVC você programa métodos e não eventos. Há também separação entre o código que interage com o usuário, o design e a camada de dados. Tudo isso parece estranho para programadores acostumados a programar eventos em formulários.

Outros programadores afirmam que têm dificuldades em aprender ASP.NET MVC. Acredito que a dificuldade no aprendizado se deve ao fato de a maioria do conteúdo estar em inglês ou em pequenos tutoriais em português. No momento em que escrevo este texto – início de junho de 2011 –, há poucos livros em português sobre ASP.NET MVC.

1.4 Quando usar o ASP.NET Web Forms

A seguir listamos alguns motivos para você continuar usando o ASP.NET Web Forms:

- Quando você precisa rapidamente de um recurso visualmente sofisticado, como os obtidos com o controle `GridView`, `DataList`, `ListView` ou `Repeater`.
- Quando há necessidade de manter o estado da página entre requisições.
- Quando for necessário ligar um controle `SqlDataSource` a uma origem de dados.
- Quando sentir saudade do recurso de arrastar e soltar do Visual Studio.

1.5 Quando usar o ASP.NET MVC

Alguns motivos para você adotar o ASP.NET MVC:

- Quando você precisa de controle total sobre o HTML.
- Quando há necessidade de unidades de teste no projeto.
- Quando a aplicação necessita de separação entre o design, o código e a camada de dados.
- Quando há obrigação de reduzir o tamanho das páginas geradas.
- Quando é preciso eliminar ou reduzir os Postbacks.
- Quando uma equipe grande desenvolve uma aplicação. Cada um pode-se dedicar a uma parte (controlador, model e view) da aplicação.
- Quando é necessário estender a aplicação constantemente.
- Quando a aplicação requer múltiplas interfaces. Por exemplo, você pode criar um view que exibe uma página HTML e outro que exibe uma página no estilo Silverlight ou em um formato para dispositivos móveis.
- Quando você não se sentir confortável desenvolvendo com formulários. Geralmente, programadores não gostam de fazer o design da aplicação.
- Quando os designs da empresa estiverem com pouco trabalho e você quer que eles criem todas as páginas manualmente, sem os característicos recursos de arrastar e soltar. Maldade, né?

1.6 Os dois ASP.NET funcionam juntos?

Sim, aproveite o melhor de cada um. Por exemplo, uma aplicação ASP.NET MVC pode exibir informações em um controle `DataList` do ASP.NET Web Forms.

1.7 Recursos do ASP.NET MVC

O ASP.NET MVC permite, entre outras coisas:

- Controle total sobre o HTML.
- Criação de URLs amigáveis.
- Clara separação entre o design, o código e a camada de dados.
- Validação no cliente e servidor.
- Definição de filtros de ação.
- Facilidade em implementar aplicações Ajax.
- Uso da sintaxe Razor.
- Uso intensivo de atributos.
- Implementação e criação de HTML helpers.
- A implementação de unidades de teste.
- Implementa o pattern MVC.

1.8 Prepare seu computador

Os exemplos deste livro podem ser desenvolvidos com o Visual Studio.

<http://www.microsoft.com/netframework>

<http://www.microsoft.com/visualstudio>

<http://www.microsoft.com/express/downloads>

Baixe e instale também o ASP.NET MVC. Acesse o website <http://www.asp.net/mvc>

Obs.: nos arquivos de exemplo, você encontra uma lista com todos os links indicados no livro. Assim, não há necessidade de digitá-los.

1.8.1 Preparando o IIS

Após a instalação dos softwares anteriores, instale o Internet Information Services (IIS).

Se você planeja usar seu computador somente para testar os exemplos deste livro, não é necessário instalá-lo. Nesse caso, use o ASP.NET Development Server, o qual é instalado com o Visual Studio.

Uma aplicação ASP.NET MVC roda no contexto da máquina local sob a conta AUTORIDADE NT\SERVIÇO DE REDE.

1.8.2 Acessando uma aplicação ASP.NET MVC

Os arquivos da aplicação web acessados pelo navegador são disponibilizados pelo servidor web – o servidor IIS. Todos os arquivos da aplicação ASP.NET MVC acessados pelo navegador devem fazer parte de um diretório virtual.

Para testar a aplicação web no seu computador, recomenda-se empregar a seguinte sintaxe:

`http://[nomedoservidor]/[DiretórioVirtual]/[método]`

Exemplo:

`http://alfredo/livro/`

ou

`http://localhost/livro/`

`http://localhost/livro/capitulo1`

de modo que `livro` é o diretório virtual, e `capitulo1`, o subdiretório. Somente o diretório-raiz, no caso `livro`, precisa ser virtual.

Obs.: navegador é o termo usado neste livro para nomear softwares de exibição de páginas de websites como o Internet Explorer, o Firefox, o Opera, o Google Chrome etc.

1.8.3 Criando um diretório virtual

- Abra o **Windows Explorer**, uma vez que um diretório virtual pode ser criado facilmente nele.
- Navegue até o diretório-raiz da aplicação web.
- Clique com o botão direito no diretório escolhido e, em seguida, clique em **propriedades**.
- Selecione a aba **Compartilhamento da web** e, em seguida, selecione **compartilhar essa pasta**.
- Defina um alias para a pasta. O alias é o nome do diretório virtual que será usado na URL do navegador. Observe a figura 1.2.

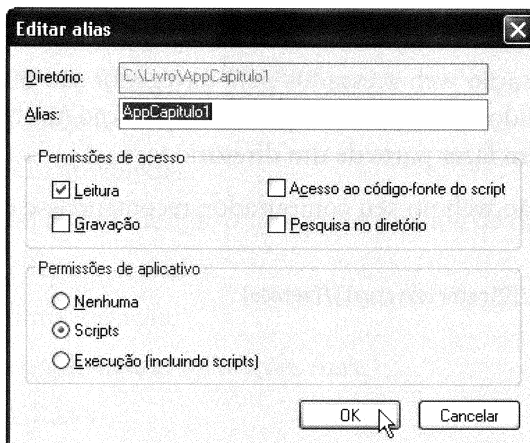


Figura 1.2 – Criando um diretório virtual.

1.9 Desenvolvendo uma nova aplicação

Inicie o Visual Studio. Acesse o menu **File** e clique em **New Project ...**.

Na caixa de diálogo **New Project** selecione **Web > Visual Studio 2012 > ASP.NET MVC 4 Web Application**.

Nomeie o projeto como AppCapitulo1. Clique em **OK**. Conforme figura 1.3.

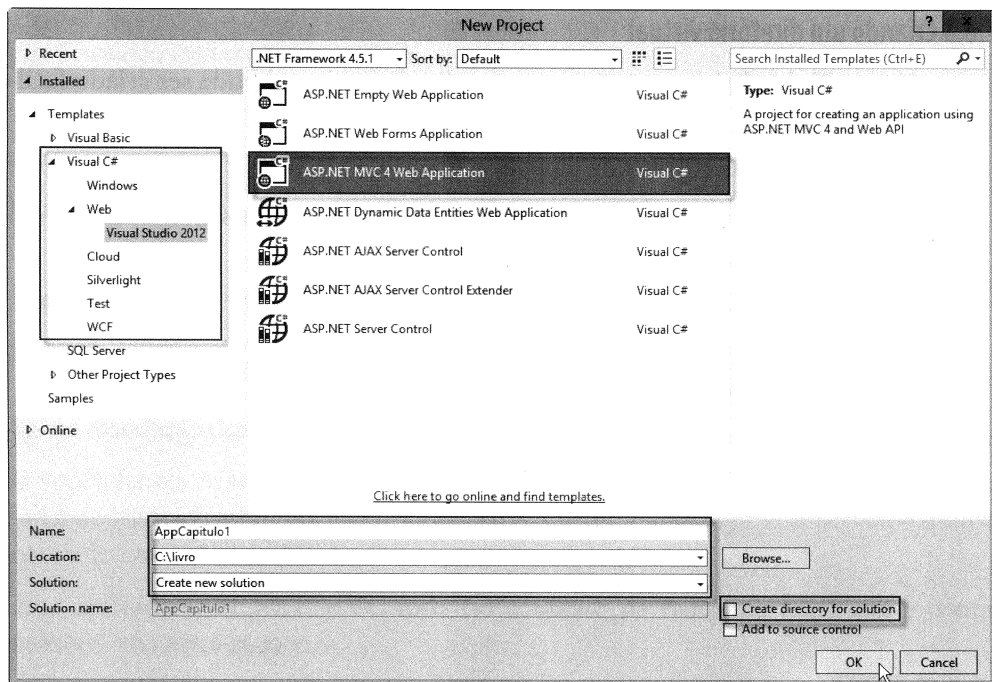


Figura 1.3 – Criando uma nova aplicação ASP.NET MVC.

Em seguida, escolha um dos dois templates para iniciar seu projeto:

- Empty contém a estrutura básica de arquivos e diretórios usados por uma aplicação ASP.NET MVC.
- Internet Application contém além dos recursos do template Empty arquivos e diretórios usados na autenticação de usuários e formatação.
- Basic contém a estrutura básica de arquivos e diretórios e também scripts jQuery, um arquivo .css e o view Error.

Na caixa de combinação **View Engine**:. Selecione o mecanismo de exibição ASPX.

Observe a figura 14.

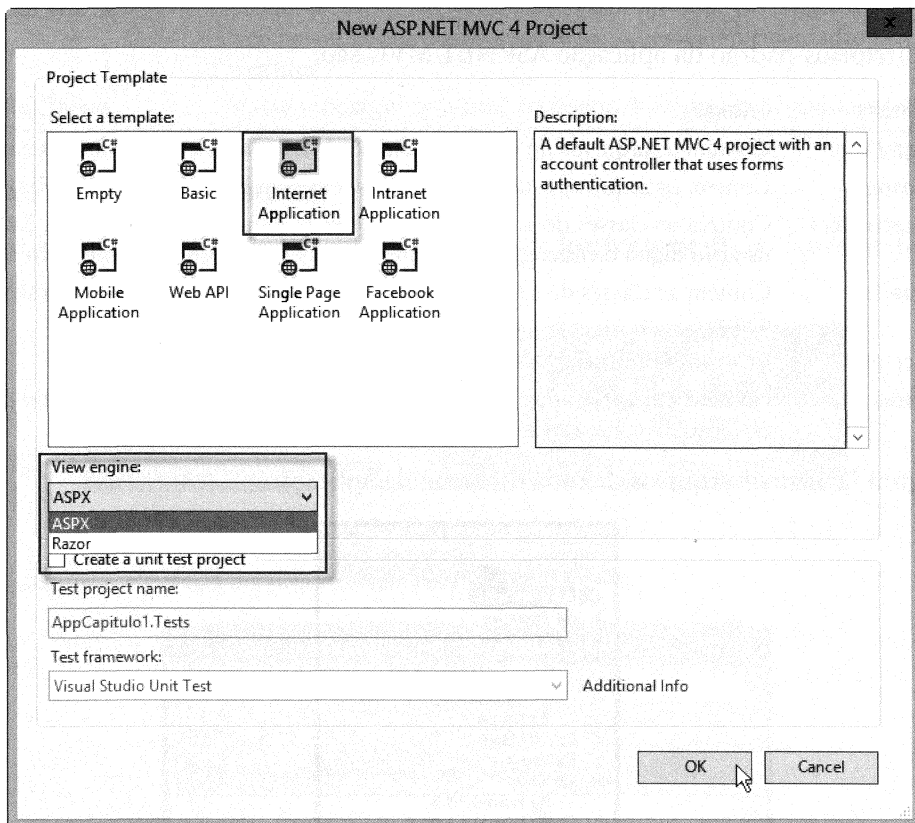


Figura 14 – Definindo o template e o mecanismo de exibição da aplicação.

Obs.: o Visual Studio Profissional ou superior permite a criação de uma unidade de teste.

Quando criamos uma nova aplicação com o template Empty, é automaticamente adicionada a seguinte estrutura de arquivos e diretórios. Observe a figura 1.5.

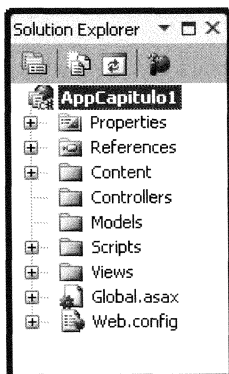


Figura 1.5 – Estrutura de arquivos e diretórios do website ASP.NET MVC.

Os diretórios-padrão da aplicação ASP.NET MVC são:

Diretório	Descrição
App_Data	Contém os arquivos do banco de dados (.MDF, .MDB), documentos xml.
Content	Contém os arquivos estáticos como imagens e arquivos CSS.
Controllers	Contém as classes do controlador. Recebe as requisições do usuário. Ex.: o usuário digita o endereço da Internet de uma página ou clique num link.
Models	Contém as classes da camada de dados. Manipula informações do banco de dados.
Scripts	Contém os arquivos JavaScript, jQuery, arquivos .js em geral.
Views	Contém em geral arquivos .aspx, .ascx, .master, .cshtml responsáveis pela exibição do conteúdo ao usuário.

A figura 1.6 lista os arquivos de cada diretório da aplicação:

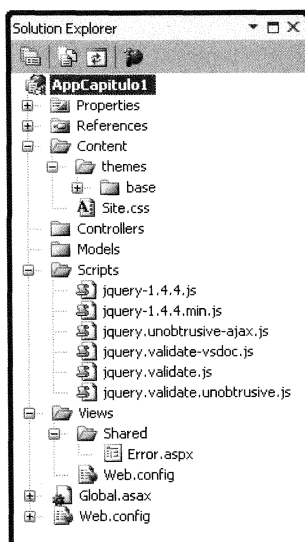


Figura 1.6 – Arquivos e diretórios da aplicação ASP.NET MVC.

A figura 1.6 exibe somente alguns arquivos do diretório Scripts.

Além dos diretórios-padrão, listados anteriormente, há diretórios com significados especiais:

Diretório	Descrição
Areas	O ASP.NET MVC cria uma nova estrutura de diretórios e arquivos. Ideal para dividir uma aplicação grande ou criar uma área restrita no website.
App_GlobalResources	Contém os arquivos de recursos globais (.resx e .resources). Esses arquivos contêm imagens, textos, arquivos etc.
App_LocalResources	Contém os arquivos de recursos locais (.resx e .resources). Esses arquivos contêm imagens, textos, outros arquivos etc.
App_Browsers	Contém os arquivos com a extensão .browser. Os arquivos .browser gerenciam as informações sobre os recursos implementados pelo navegador.
App_Themes	Contém os arquivos .skin e .css responsáveis pela aparência da aplicação.

1.10 Executando a aplicação

Para executar a aplicação, pressione **F5** ou acesse **Start Debugging (F5)** na barra de ferramentas do Visual Studio. Observe a figura 1.7:

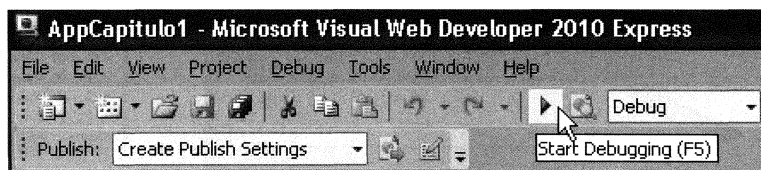


Figura 1.7 – Executando a aplicação.

Em seguida, é gerada a saída exibida na figura 1.8.



Figura 1.8 – Saída exibe uma página inválida.

Bem, esse erro acontece porque não temos nenhuma informação ou página para exibir no navegador. Para resolver esse problema adicionamos um controlador ao projeto.

1.11 Controladores

Os controladores MVC manipulam e respondem às entradas e interações do usuário e também interagem com as outras partes da aplicação, como views e models.

Para adicionar um novo controlador à aplicação web, acesse a janela **Solution Explorer**. Clique com o botão direito do mouse no diretório **Controllers**. Em seguida, aponte para **Add** e clique em **Controller...**. Na caixa de diálogo **Add Controller** digite o nome do controlador – **HomeController**. Observe a figura 1.9.

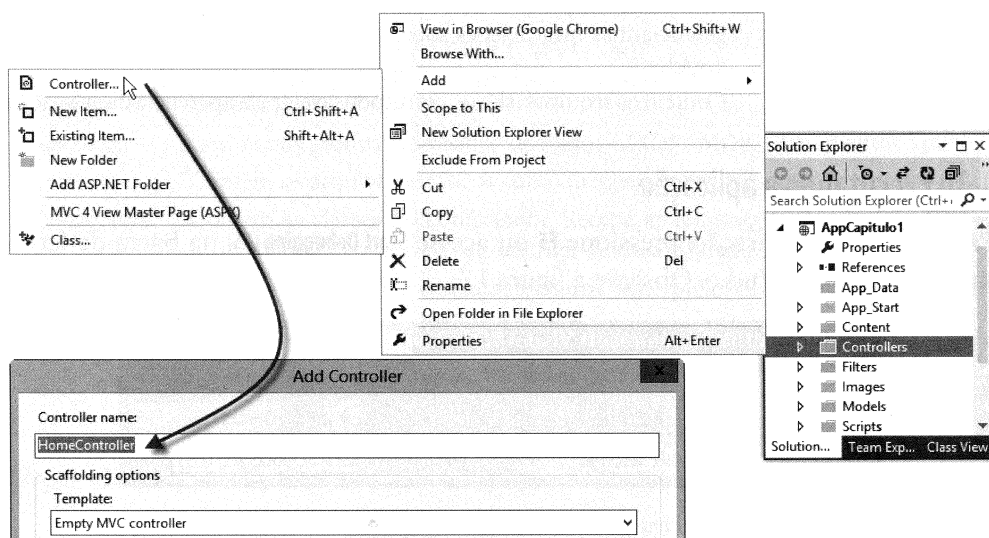


Figura 1.9 – Adicionando um novo controlador.

O ASP.NET MVC requer controladores com o sufixo **Controller**. Exemplo: **HomeController**, **DetailsController**, **EditController** etc. O prefixo é usado nas entradas e interações do usuário.

Exemplo:

`http://localhost:1573/Home`

`http://localhost:1573/Home/Details/5`

`http://localhost:1573/Home/Edit/5`

Obs.: os action method são denominados também métodos de controlador.

Quando clicamos no botão **Add** da caixa de diálogo **Add Controller**, o Visual Studio acrescenta o arquivo `HomeController.cs` ao diretório `Controllers` e o seguinte código:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace AppCapitulo1.Controllers {
    public class HomeController : Controller {
        // GET: /Home/
        public ActionResult Index() {
            return View();
        }
    }
}
```

Pressione **F5**. A aplicação retorna novamente uma página de erro, pois ainda não associamos nenhum view – responsável pela exibição do conteúdo ao usuário.

Para resolver o problema, substitua o trecho de código:

```
public ActionResult Index() {
    return View();
}
```

Por:

```
public ActionResult Index() {
    return Content("<h1>Olá mundo!</h1>");
}
```

Se preferir use:

```
public string Index() {
    return "<h1>Olá mundo!</h1>";
}
```

Ou:

```
public void Index() {
    Response.Write("<h1>Olá mundo!</h1>");
}
```

Pressione **F5** novamente. O resultado vemos na figura 1.10.

O mesmo resultado é obtido quando digitamos:

`http://localhost:1029/`

`http://localhost:1029/Home`

`http://localhost:1029/Home/index`

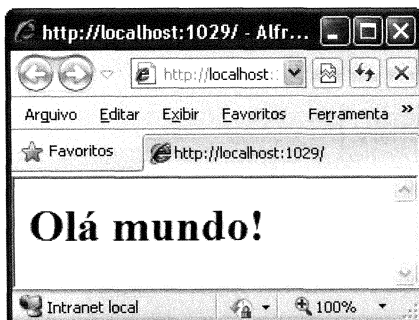


Figura 1.10 – Nosso primeiro exemplo em ação.

O ASP.NET MVC usa roteamento de URLs e as regras são registradas no método `RegisterRoutes` dentro do arquivo `Global.asax`:

```
public static void RegisterRoutes(RouteCollection routes) {  
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");  
    routes.MapRoute(  
        "Default", // Route name  
        "{controller}/{action}/{id}", // URL with parameters  
        new { controller = "Home", action = "Index", id = UrlParameter.Optional }  
    );  
}
```

`controller` é o nome da classe do controlador. `action` é o nome do método invocado. E `id` é o parâmetro opcional embutido na URL e usado para passar argumentos para o método.

O formato de URL `{controller}/{action}/{id}` registrado no `Global.asax` permite as seguintes variações de URLs:

`http://localhost:1029/`

`http://localhost:1029/Home`

`http://localhost:1029/Home/index`

`http://localhost:1029/Home/Create`

`http://localhost:1029/Home/Details/5`

`http://localhost:1029/Home/Edit/25`

Obs.: baixe os arquivos de exemplo no site da Novatec Editora, assim você não precisa digitar grandes quantidades de código.

1.11.1 Método com parâmetros

Acrescente o método Details à classe HomeController:

```
public ActionResult Details(int id) {  
    return Content("<h1>Olá mundo com id = " + id + "</h1>");  
}
```

Exemplo:

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Web;  
using System.Web.Mvc;  
  
namespace AppCapitulo1.Controllers {  
    public class HomeController : Controller {  
        public ActionResult Index() {  
            return Content("<h1>Olá mundo!</h1>");  
        }  
  
        public ActionResult Details(int id) {  
            return Content("<h1>Olá mundo com id = " + id + "</h1>");  
        }  
    }  
}
```

Quando pressionamos **F5** ou digitamos no navegador a URL:

<http://localhost:NúmeroDaPorta/Home/Details/5>

É retornada a página exibida na figura 1.11:

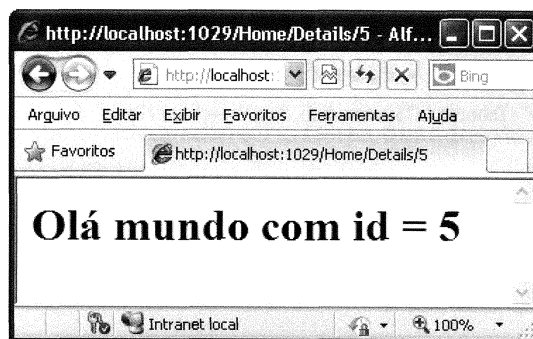


Figura 1.11 – URL com parâmetros.

Obs.: 1029 é o número da porta usada pelo ASP.NET Development Server. Esse valor não é fixo. Ao pressionar **F5** no seu computador o número da porta será outro. Altere o número da porta ou coloque a aplicação num diretório virtual e digite: <http://localhost/nomediretóriovirtual/Home/Details/5>

1.12 Master pages

Para exibir elementos comum a todas as páginas, use master pages. Uma master page pode facilmente exibir menus, logos, banners, e seu layout pode conter textos estáticos, elementos HTML, imagens e web server controls.

Para acrescentar uma master page a um projeto ASP.NET MVC, siga os passos descritos a seguir:

Clique com o botão direito do mouse no diretório /Views/Shared. Selecione **Add** e, em seguida, aponte para **New Item....** Observe a figura 1.12:

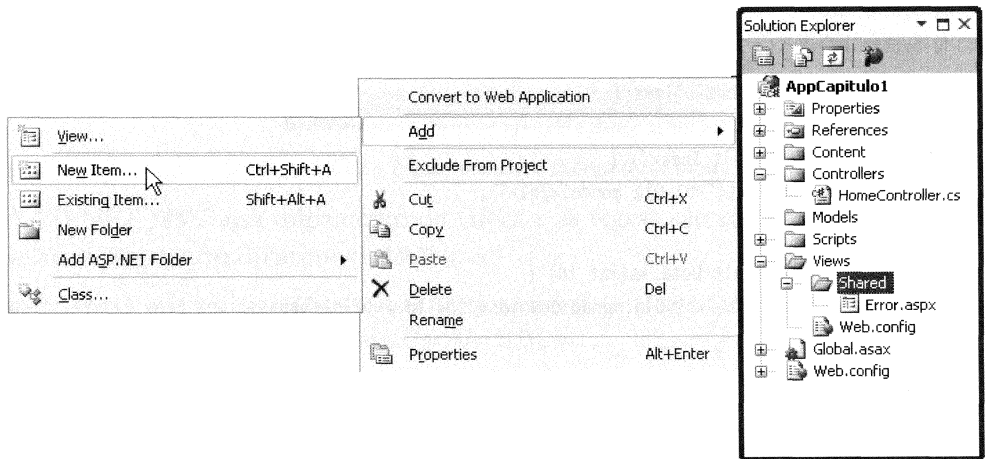


Figura 1.12 – Acrescentando uma nova master page.

Na caixa de diálogo **Add New Item** selecione **MVC 4 View Master Page (ASPX)**. Nomeie como Site.Master. Clique em **Add**. Observe a figura 1.13.

O arquivo Site.Master contém o seguinte código:

```
<%@ Master Language="C#" Inherits="System.Web.Mvc.ViewMasterPage" %>
<!DOCTYPE html>
<html>
  <head runat="server">
    <title><asp:ContentPlaceholder ID="TitleContent" runat="server" /></title>
  </head>
  <body>
    <div>
      <asp:ContentPlaceholder ID="MainContent" runat="server">
        </asp:ContentPlaceholder>
      </div>
    </body>
  </html>
```

Um controle `ContentPlaceholder` define regiões da página que não têm conteúdo comum a todas as páginas. Por exemplo, os produtos mais vendidos.

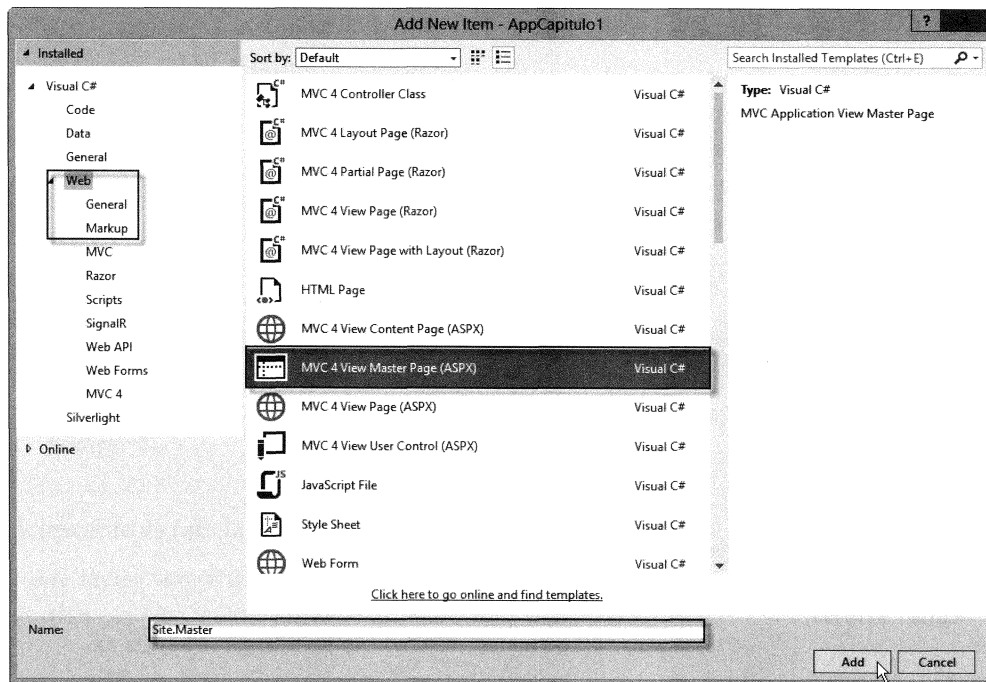


Figura 1.13 – Caixa de diálogo Add New Item.

Para formatar o website usamos folhas de estilo CSS. Acrescente à tag `head` da master page uma referência para a tag `link`:

```
<head runat="server">
    <link href="/Content/Site.css" rel="Stylesheet" type="text/css"/>
    <title><asp:ContentPlaceholder ID="TitleContent" runat="server"/></title>
</head>
```

O arquivo `Site.css` é adicionado ao diretório `Content` quando criamos um novo projeto ASP.NET MVC e possui os seguintes seletores CSS:

```
body {
    font-size: 75%;
    font-family: Verdana, Tahoma, Arial, "Helvetica Neue", Helvetica, Sans-Serif;
    color: #232323;
    background-color: #fff;
}

/* Estilos para formulários
-----*/
fieldset {
    border: 1px solid #ddd;
```

```
padding:0 1.4em 1.4em 1.4em;
margin:0 0 1.5em 0;
}
legend {
    font-size:1.2em;
    font-weight: bold;
}

textarea {
    min-height: 75px;
}

.editor-label {
    margin: 1em 0 0 0;
}

.editor-field {
    margin:0.5em 0 0 0;
}

/* Estilos para validação
-----*/
.field-validation-error {
    color: #ff0000;
}

.field-validation-valid {
    display: none;
}

.input-validation-error {
    border: 1px solid #ff0000;
    background-color: #ffebee;
}

.validation-summary-errors {
    font-weight: bold;
    color: #ff0000;
}

.validation-summary-valid {
    display: none;
}
```

1.12.1 Configurando a master page

Inicialmente, usamos a master page para formatar o topo das páginas da nossa aplicação ASP.NET MVC.

Adicione ao arquivo Site.css o trecho de código a seguir:

```

body {
    font-size: 75%;
    font-family: Verdana, Tahoma, Arial, "Helvetica Neue", Helvetica, Sans-Serif;
    color: #232323;
    background-color: #fff;
    margin-left: 0;
    margin-right: 0;
    margin-top: 0;
    margin-bottom: 0;
}

table {
    border-collapse: collapse;
    border: 0;
}

td {
    padding: 0;
}

```

Acrescente as tabelas a seguir ao arquivo Site.Master:

```

<table style="text-align: center; border: 0; width: 100%">
    <tr>
        <td style="background-color: #006486; height: 5px;" colspan="4">
        </td>
    </tr>
    <tr>
        <td style="height: 65px; width: 15px;">
            <strong>LOGO Website</strong>
        </td>
        <td style="text-align: right;">
            <table style="width: 100%; border: 0px;">
                <tr>
                    <td style="width: 100%; text-align: right; font-size: 40px; font-family: Arial;">
                    </td>
                </tr>
            </table>
        </td>
    </tr>
</table>
<table style="width: 100%; text-align: center; border: 0;">
    <tr>
        <td style="background-color: #006486; height: 5px;">
        </td>
    </tr>
</table>

```

Exemplo:

```

<%@ Master Language="C#" Inherits="System.Web.Mvc.ViewMasterPage" %>
<!DOCTYPE html>
<html>
  <head runat="server">
    <link href="/Content/Site.css" rel="Stylesheet" type="text/css" />
    <title>
      <asp:ContentPlaceholder ID="TitleContent" runat="server" />
    </title>
  </head>
  <body>
    <div>
      <table style="text-align: center; border: 0; width: 100%">
        <tr>
          <td style="background-color: #006486; height: 5px;" colspan="4">
          </td>
        </tr>
        <tr>
          <td style="height: 65px; width: 15px;">
            <strong>LOGO Website</strong>
          </td>
          <td style="text-align: right;">
            <table style="width: 100%; border: 0px;">
              <tr>
                <td style="width: 100%; text-align: right; font-size: 40px;
                  font-family: Arial;">
                </td>
              </tr>
            </table>
          </td>
        </tr>
      </table>
      <table style="width: 100%; text-align: center; border: 0;">
        <tr>
          <td style="background-color: #006486; height: 5px;">
          </td>
        </tr>
      </table>
      <asp:ContentPlaceholder ID="MainContent" runat="server">
    </asp:ContentPlaceholder>
    </div>
  </body>
</html>

```


Para aplicar a master page em um view, basta adicionar o atributo `MasterPageFile` à diretiva `@ page`:

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
    Inherits="System.Web.Mvc.ViewPage<dynamic>" %>
```

A figura 1.14 mostra a master page `Site.Master` em ação:

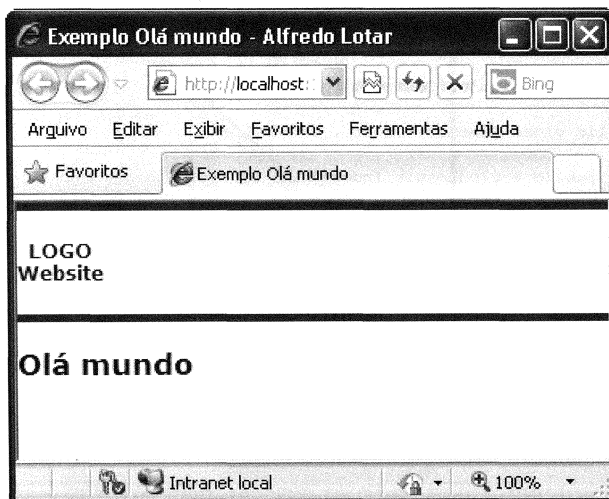


Figura 1.14 – Cabeçalho da página formatada pela master page.

Obs.: baixe os arquivos de exemplo no site da Editora Novatec e, em seguida, copie o código anteriormente mencionado.

1.13 Views

View ou exibição – é responsável pela apresentação do conteúdo HTML da página. Geralmente, é definida por intermédio de arquivos `.aspx`, `.ascx`, `.master`, `.cshtml`.

Criar uma exibição a partir de um método de controlador MVC. É o modo mais fácil, rápido e simples.

Clique com o botão direito do mouse no nome do método e, em seguida, clique em **Add View...**. Observe a figura 1.15.

Na caixa de diálogo **Add View** defina o nome do view e também o mecanismo de exibição – ASPX no nosso caso. Marque a caixa de seleção com a legenda "Use a layout or master page:" e determine o nome da master page. Observe a figura 1.16.

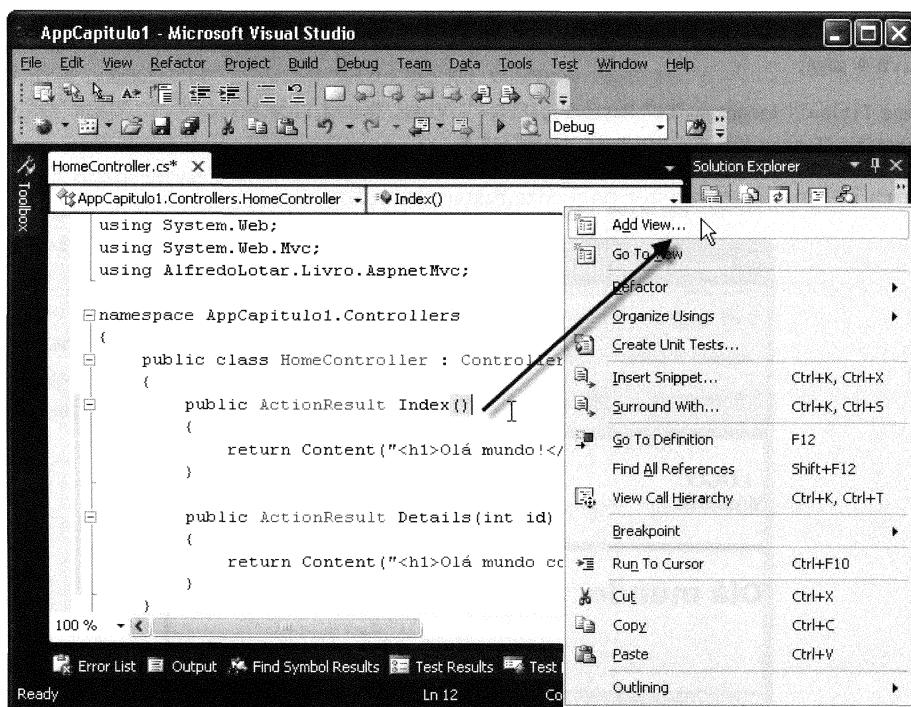


Figura 1.15 – Adicionando um novo View.

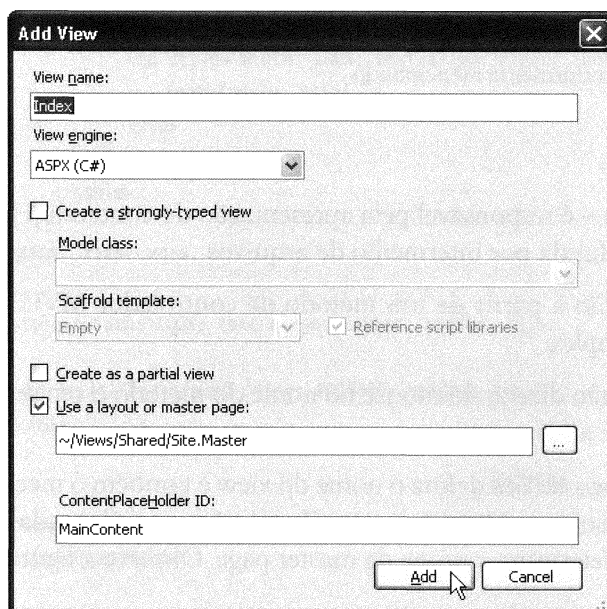


Figura 1.16 – Caixa de diálogo Add View.

O novo view é acrescentado ao subdiretório `Home` do diretório `Views`. Observe a figura 1.17:

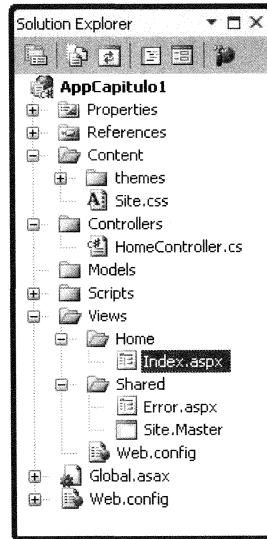


Figura 1.17 – View `Index.aspx`.

Quando abrimos o arquivo `Index.aspx` vemos o seguinte conteúdo:

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
    Inherits="System.Web.Mvc.ViewPage<dynamic>" %>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
    Index
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
    <h2>Index</h2>
</asp:Content>
```

1.13.1 Exibindo informações num view

Normalmente, passamos informações de um método de controlador para um view por intermédio da propriedade `ViewBag`. Cujas sintaxe é:

```
ViewBag.QualquerNome = Conteúdo;
```

Exemplo:

```
ViewBag.Nome = "Alfredo Lotar";
ViewBag.Editora = "Novatec";
ViewBag.Idade = 37;
ViewBag.Data = DateTime.Now;
```

Para exibir a mensagem "ASP.NET MVC!" na tela, acrescentamos ao método `Index` da classe `HomeController` a seguinte linha:

```
ViewBag.Mensagem = "ASP.NET MVC!";
```

Exemplo:

```
using System.Web.Mvc;

namespace AppCapitulo1.Controllers {
    public class HomeController : Controller {
        public ActionResult Index() {
            ViewBag.Mensagem = "ASP.NET MVC!";
            return View();
        }

        public ActionResult Details(int id) {
            return View();
        }
    }
}
```

Em seguida, adicionamos a linha a seguir:

```
<%=ViewBag.Mensagem%>
```

Ao arquivo `Index.aspx`. Exemplo:

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
    Inherits="System.Web.Mvc.ViewPage<dynamic>" %>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
    Index
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
    <h2><%=ViewBag.Mensagem%></h2>
</asp:Content>
```

A saída no ASP.NET MVC é gerada pela expressão inline: `<%= expressão %>`

Obs.: as versões 1 e 2 do ASP.NET MVC usam a expressão `<%= expressão %>` em vez de `<%= expressão %>`.

1.13.2 Acrescentando um link ao view

O modo mais usado para linkar páginas no ASP.NET MVC é por intermédio do método `ActionLink`. Exemplo:

```
<%=Html.ActionLink("Volta para página inicial", "Index") %>
```

O primeiro parâmetro do método `ActionLink` define o texto do link; o segundo, o nome do método que deve ser executado; o terceiro parâmetro opcional embutido na URL é usado para passar argumentos para o método.

Para exemplificar, altere o método `Details` da classe `HomeController` para:

```
public ActionResult Details(int id) {  
    ViewBag.Mensagem = "Detalhes id = " + id;  
    return View();  
}
```

Em seguida, associe um novo view para o método `Details`. Observe a figura 1.18.

Na página `Details.aspx` acrescente as linhas a seguir:

```
<h2>  
    <%=ViewBag.Mensagem%>  
</h2>  
<%=Html.ActionLink("Volta para página inicial", "Index") %>
```

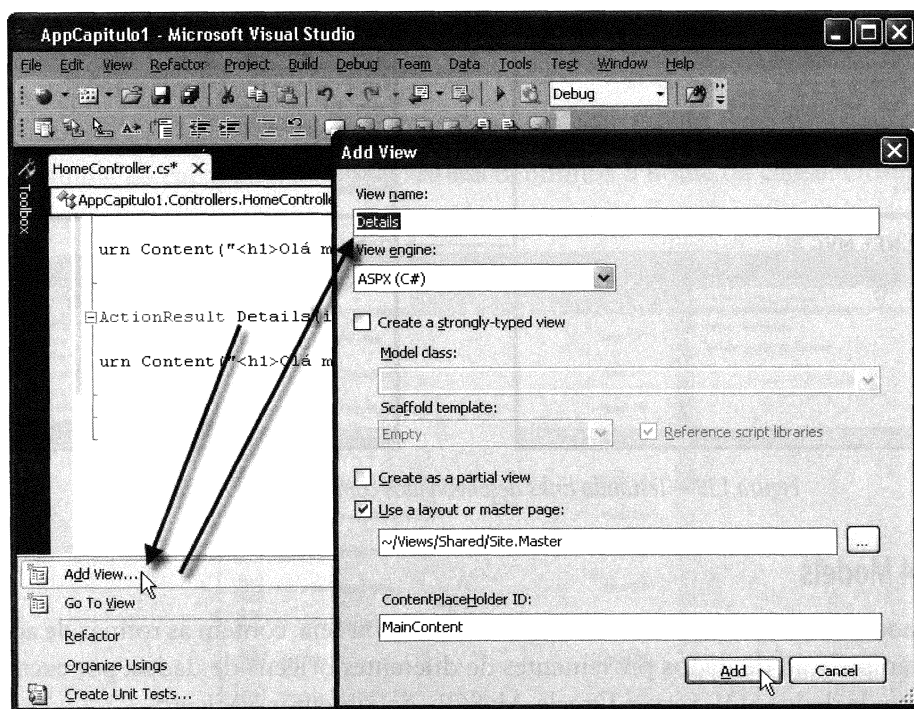


Figura 1.18 – Criando um novo View.

Exemplo:

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
    Inherits="System.Web.Mvc.ViewPage<dynamic>" %>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
    Detalhes
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
    <h2>
        <%=ViewBag.Mensagem%>
    </h2>
    <%=Html.ActionLink("Volta para página inicial", "Index") %>
</asp:Content>
```

E no arquivo Index.aspx acrescente a linha:

```
<%=Html.ActionLink("Página detalhes", "Details", new {id = 5}) %>
```

Navegue entre as páginas e veja os links funcionando. Observe a figura 1.19:

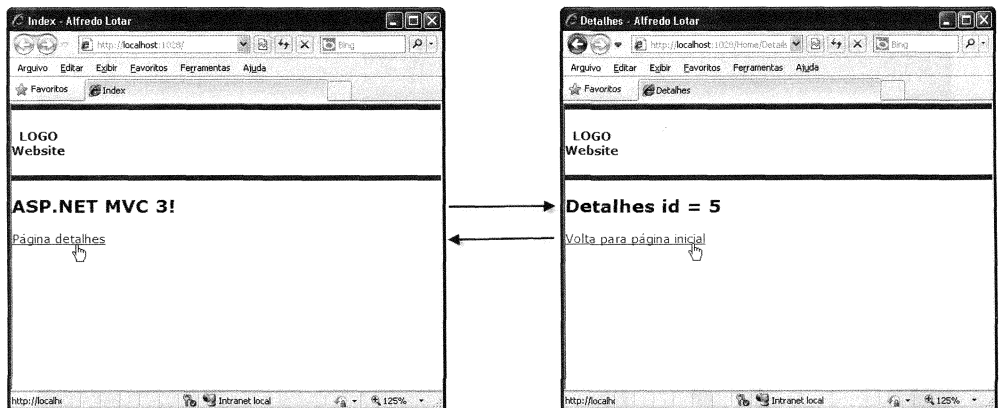


Figura 1.19 – Testando links desenvolvidos com o método ActionLink.

1.14 Models

O model contém as classes da camada de dados, ou seja, contém as rotinas de acesso e manipulação de dados provenientes de diferentes origens de dados, por exemplo, banco de dados SQL Server, Oracle, MySQL, documentos XML etc.

Os arquivos da camada de dados são armazenados no diretório Models.

Para facilitar a manutenção, os testes e evitar a duplicação de códigos, divida, por exemplo, a camada de dados em: um ou mais objetos Entity Framework e os manipule por intermédio de uma classe e uma interface. No capítulo 4 abordaremos o assunto novamente.

Ao clicar no botão **Add** da caixa de diálogo **Add New Item** é acrescentado ao diretório `Models` o arquivo `Cidades.cs` com o seguinte código:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

namespace AppCapitulo1.Models {
    public class Cidades {
    }
}
```

O passo seguinte é alterar o código gerado. Alteramos o nome da namespace de:

```
namespace AppCapitulo1.Models {
```

Para:

```
namespace AlfredoLotar.Livro.AspNetMvc {
```

E acrescentamos três propriedades:

```
public int CidadeID { get; set; }
public string Nome { get; set; }
public string Estado { get; set; }
```

A seguir temos o código da classe `Cidades` com as alterações:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

namespace AlfredoLotar.Livro.AspNetMvc {
    public class Cidades {
        public int CidadeID { get; set; }
        public string Nome { get; set; }
        public string Estado { get; set; }
    }
}
```

Antes de acessar os membros da classe `Cidades` é preciso compilá-la. Observe a figura 1.22:

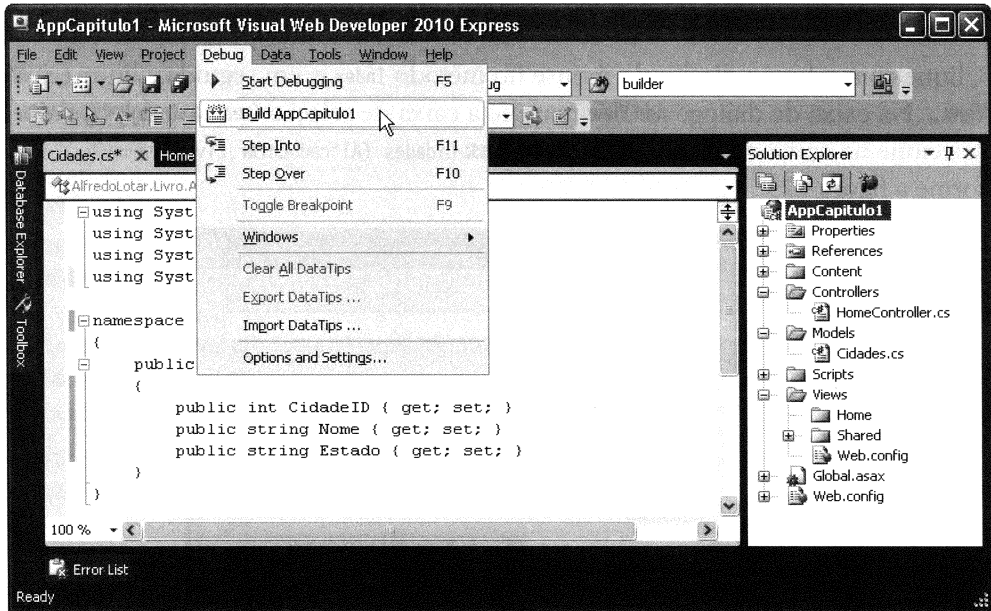


Figura 1.22 – Compilando a aplicação a partir do menu Debug.

1.14.2 Criando uma instância da classe Cidades

O próximo passo do nosso exemplo é a criação de uma instância da classe `Cidades` em um método de controlador qualquer. Nomeamos o método como `Index`. Se preferir, use qualquer outro nome.

Exemplo:

```
using System.Web.Mvc;
using AlfredoLotar.Livro.AspnetMvc;

namespace AppCapitulo1.Controllers {
    public class HomeController : Controller {
        public ActionResult Index() {
            var model = new Cidades() {
                CidadeID=1,
                Nome="São Paulo",
                Estado="SP"
            };
            return View(model);
        }
    }
}
```

1.14.3 Acrescentando um view

Clique com o botão direito do mouse no método `Index` e, em seguida, clique em **Add View...** Na caixa de diálogo **Add View** marque a caixa de seleção **Create a strongly-typed view** e selecione na caixa de combinação **Model Class:** `Cidades (AlfredoLotar.Livro.AspNetMvc)`, conforme a figura 1.23:

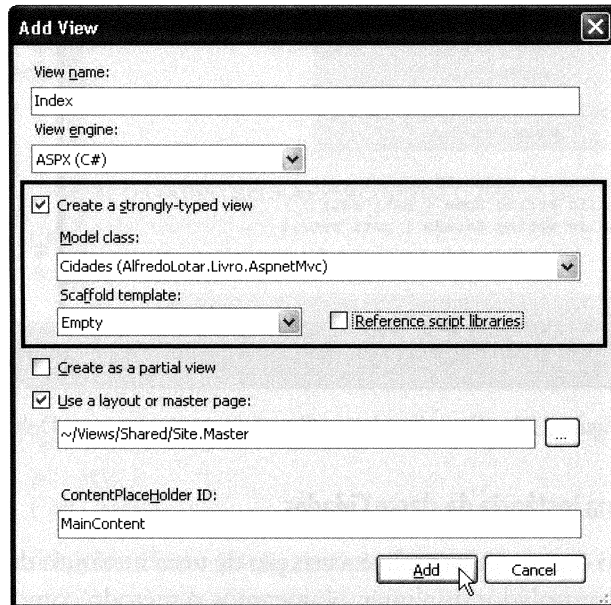


Figura 1.23 – Caixa de diálogo Add View.

Após clicar em **Add** é adicionado o arquivo `Index.aspx` ao diretório `Views\Home` com o seguinte código:

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
    Inherits="System.Web.Mvc.ViewPage<AlfredoLotar.Livro.AspNetMvc.Cidades>" %>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
    Index
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
    <h2>Index</h2>
</asp:Content>
```

A classe usada pelo view é declarada na diretiva `@ Page`:

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
    Inherits="System.Web.Mvc.ViewPage<AlfredoLotar.Livro.AspNetMvc.Cidades>" %>
```

No view, as propriedades da classe `Cidades` são acessadas por intermédio do objeto `Model`:

```
<%:Model.CidadeID %>  
<%:Model.Nome %>  
<%:Model.Estado %>
```

Exemplo:

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"  
    Inherits="System.Web.Mvc.ViewPage<AlfredoLotar.Livro.AspNetMvc.Cidades>" %>  
  
<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">  
    Cidades  
</asp:Content>  
  
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">  
    <h2>Cidades</h2>  
    <%:Model.CidadeID %>  
    <%:Model.Nome %>  
    <%:Model.Estado %>  
</asp:Content>
```

A saída no navegador você vê na figura 1.24:

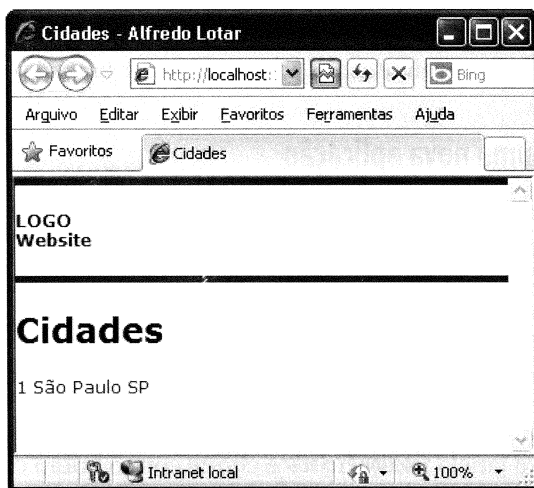


Figura 1.24 – Exemplo em ação.

Controladores

O controlador (controller de uma aplicação ASP.NET MVC) contém as classes e os métodos de controlador. É por meio dessas classes e métodos que são processadas as requisições do usuário. Requisição pode ser um clique em um link no website, a digitação e acesso a um endereço de Internet no navegador ou enviar o conteúdo de um formulário.✓

Nas classes e nos métodos de controlador aplicamos atributos para limitar a operação executada, para nomear métodos de controlador, definir cache, restringir o acesso a determinados usuários ou páginas com SSL.

Os métodos de controlador retornam diferentes tipos, por exemplo, notações JSON, conteúdo JavaScript e de arquivos (.pdf, .doc etc.), view, partial view, entre outros.

2.1 Desenvolvendo uma nova aplicação

Inicie o Visual Studio. Acesse o menu **File** e clique em **New Project...** Na caixa de diálogo **New Project** selecione **Web > Visual Studio 2012 > ASP.NET MVC 4 Web Application**. Nomeie o projeto como AppCapítulo2. Clique em **OK**.

Acrescente ao diretório **Models** o arquivo **Cidades.cs**. Siga os passos descritos no tópico “1.14.1 Acrescentando uma nova classe”.

Como já mencionamos no capítulo anterior, para adicionar um novo controlador à aplicação web, acesse a janela **Solution Explorer** e clique com o botão direito do mouse no diretório **Controllers**. Em seguida, aponte para **Add** e clique em **Controller...** Na caixa de diálogo **Add Controller** digite o nome do controlador – **HomeController**. Observe a figura 2.1.

Quando marcamos a caixa de seleção **Add action methods for Create, Update, Delete, and Details scenarios**, automaticamente, criamos métodos para inserir, editar e excluir informações. **Details** é usado para exibir informações específicas de um registro.

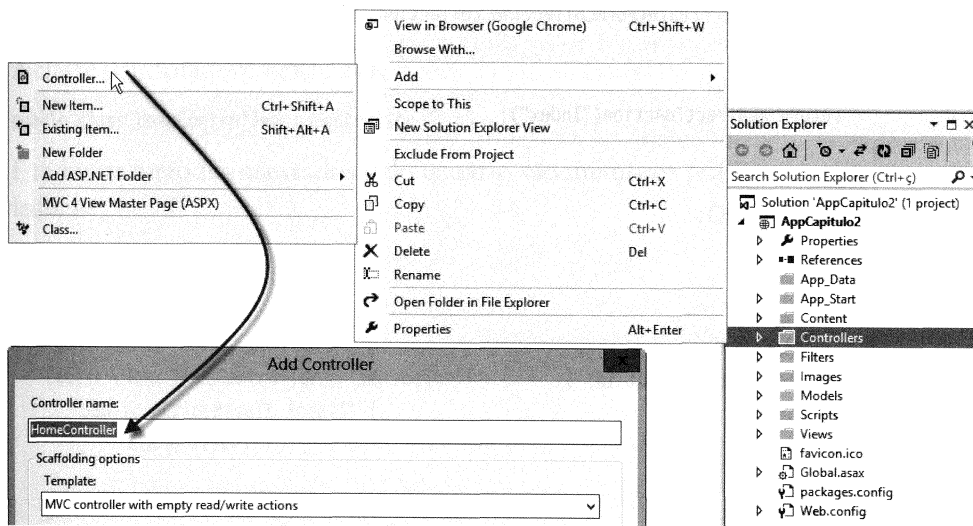


Figura 2.1 – Adicionando um novo controlador.

Ao clicar em **Add**, o código a seguir é inserido no arquivo HomeController.cs:

```
using System.Web.Mvc;

namespace AppCapitulo2.Controllers {
    public class HomeController : Controller {
        // GET: /Home/
        public ActionResult Index() {
            return View();
        }

        // GET: /Home/Details/5
        public ActionResult Details(int id) {
            return View();
        }

        // GET: /Home/Create
        public ActionResult Create() {
            return View();
        }

        // POST: /Home/Create
        [HttpPost]
```

```
public ActionResult Create(FormCollection collection) {
    try {
        // Código aqui
        return RedirectToAction("Index");
    }
    catch {
        return View();
    }
}

// GET: /Home/Edit/5
public ActionResult Edit(int id) {
    return View();
}

// POST: /Home/Edit/5
[HttpPost]
public ActionResult Edit(int id, FormCollection collection) {
    try {
        // Código aqui
        return RedirectToAction("Index");
    }
    catch {
        return View();
    }
}

// GET: /Home/Delete/5
public ActionResult Delete(int id) {
    return View();
}

// POST: /Home/Delete/5
[HttpPost]
public ActionResult Delete(int id, FormCollection collection) {
    try {
        // Código aqui
        return RedirectToAction("Index");
    }
    catch {
        return View();
    }
}
}
```

2.2 Classe controladora

A classe controladora herda as funções da classe `Controller`.

```
public class HomeController : Controller {...}
```

E todos os métodos acessados pelo usuário são membros públicos da classe controladora.

Exemplo:

```
using System.Web.Mvc;
namespace AppCapitulo2.Controllers {
    public class HomeController : Controller {
        public ActionResult Index() {
            return View();
        }
    }
}
```

No ASP.NET Web Forms você acessa páginas:

<http://www.novatec.com.br/default.aspx>

No ASP.NET MVC você invoca métodos quando digita a URL no navegador ou clica em um link:

<http://www.novatec.com.br/Home>

<http://www.novatec.com.br/Home/Details/5>

A URL tem o seguinte formato:

<http://www.novatec.com.br/{controller}/{action}/{id}>

A seguir listamos os métodos usados pela maioria das aplicações ASP.NET MVC:

Ação	URL de exemplo
Index	Home/Index
Create	Home/Create
Details	Home/Details/10
Insert	Home/Insert
Edit	Home/Edit/10
Update	Home/Update/10
Destroy	Home/Destroy/10
Delete	Home/Delete/10
Login	Home/Login
Logout	Home/Logout
Authenticate	Home/Authenticate

Você pode criar métodos com qualquer outro nome ou usar métodos com nomes em português. Exemplo: *Indice*, *Criar*, *Detalhes*, *Inserir*, *Editar*, *Atualizar*, *Confirmar*, *Excluir*, *Acessar*, *Sair*, *Autenticar*.

Os métodos de ação da classe controladora sempre retornam *ActionResult*:

```
public ActionResult Index() {  
    return View();  
}
```

E, quando parece que não retornam *ActionResult*, retornam uma classe derivada, herda de *ActionResult*.

Exemplo:

```
public string Index() {  
    return "<h1>Olá mundo!</h1>";  
}
```

O método *Index* retorna uma string que é convertida para a classe derivada *ContentResult* de *ActionResult*:

```
public class ContentResult : ActionResult
```

2.2.1 Parâmetros

No ASP.NET Web Forms, para capturar as informações postadas por uma página, é preciso escrever código específico para extrair os parâmetros:

```
int resultado;  
if (Int32.TryParse(Request.QueryString["id"], out resultado))  
{  
}
```

Ou o conteúdo de um campo de formulário:

```
string nome = Request.Form["txtNome"].ToString();
```

Agora, no ASP.NET MVC existe um mecanismo denominado *Model Binders*. ✓

Model binder no ASP.NET MVC provê um modo simples de mapear informações postadas para um tipo de dados do .NET Framework. Esse tipo é passado para um método de controlador como um parâmetro.

A classe *DefaultModelBinder* mapeia os seguintes tipos de objetos para uma requisição do navegador:

- Tipos primitivos, como *String*, *Double*, *Decimal*, ou objetos *DateTime*.
- Classes, como *Person*, *Categorie*, ou *Product*.
- Coleções, como *ICollection<T>*, *IList<T>*, ou *IDictionary<TKey, TValue>*.

Por exemplo, os campos do formulário a seguir:

```
<form action="/Home/Create" method="post">
  <div class="editor-field">
    <input name="CategoryName" type="text" value="" />
  </div>
  <div class="editor-field">
    <textarea cols="20" name="Description" rows="2" style="width: 250px;"></textarea>
  </div>
  <p>
    <input type="submit" value="Create" />
  </p>
</form>
```

São mapeados para o método de controlador Create:

```
public ActionResult Create(string CategoryName, string Description) {
    // Restante do código aqui
    return View();
}
```

As propriedades da classe Product:

```
public class Product {
    public int ProductID { get; set; }
    public string ProductName { get; set; }
    public int SupplierID { get; set; }
    public int CategoryID { get; set; }
    public decimal UnitPrice { get; set; }
    public short UnitsInStock { get; set; }
    public short UnitsOnOrder { get; set; }
    public short ReorderLevel { get; set; }
    public bool Discontinued { get; set; }
}
```

Também podem ser mapeadas:

```
public ActionResult Create(Product p) {
    // Restante do código aqui
    return View();
}
```

O model binder cria uma nova instância da classe Product e preenche as propriedades da classe com os valores passados pela requisição do navegador. Assim não precisamos escrever código específico para extrair informações passadas por uma requisição.

Por exemplo, as linhas a seguir inserem um novo produto na tabela Product do banco de dados Northwind:

```
NorthwindBD db = new NorthwindBD();  
[HttpPost]  
public ActionResult Create(Product p) {  
    try {  
        if (!ModelState.IsValid) return View();  
        db.AddToProducts(p);  
        db.SaveChanges();  
        return RedirectToAction("Index");  
    }  
    finally {  
        if (db != null) db.Dispose();  
    }  
}
```

Não é preciso escrever código específico para capturar os dados digitados no formulário. Esse trabalho é realizado pelo model binder.

Obs.: os exemplos deste livro usam o banco de dados Northwind. Para criá-lo execute o arquivo CriarNorthwindBD.bat localizado nos arquivos de exemplo deste livro. Ou execute os arquivos Northwind.sql e Createdatabase.sql no SQL Server 2008 Management Studio.

Resultado semelhante você obtém quando usa a classe `FormCollection` e o método `TryUpdateModel`.

Exemplo:

```
NorthwindBD db = new NorthwindBD();  
[HttpPost]  
public ActionResult Create(FormCollection collection) {  
    try {  
        if (!ModelState.IsValid) return View();  
        var p = new Product();  
        this.TryUpdateModel(p, collection.ToValueProvider());  
        db.AddToProducts(p);  
        db.SaveChanges();  
        return RedirectToAction("Index");  
    }  
    finally {  
        if (db != null) db.Dispose();  
    }  
}
```

O método `TryUpdateModel` atribui os campos do formulário para uma instância da classe `Product`:

```
var p = new Product();  
this.TryUpdateModel(p, collection.ToValueProvider());
```

ModelState.IsValid valida os campos do formulário:

```
if (!ModelState.IsValid) return View();
```

Obs.: a criação de formulários é abordada no capítulo 7 e a verificação e validação de informações, no capítulo 8.

2.2.1.1 Valor-padrão de parâmetros

Você pode definir valores-padrão para parâmetros por intermédio do atributo DefaultValue:

```
public ActionResult Details([DefaultValue(5)]int id) {  
    ViewBag.Mensagem = "Detalhes id = " + id;  
    return View();  
}
```

Ou diretamente:

```
public ActionResult Details(int id=5) {  
    ViewBag.Mensagem = "Detalhes id = " + id;  
    return View();  
}
```

Pode permitir que um parâmetro aceite valores nulos:

```
public ActionResult Details(int? id) {}
```

E fazer verificações:

```
public ActionResult Details(int? id) {  
    if (!id.HasValue)  
        return RedirectToAction("Index");  
    ViewBag.Mensagem = "Detalhes id = " + id;  
    return View();  
}
```

No exemplo, o método RedirectToAction redireciona o usuário para a página inicial.

Obs.: importe a namespace System.ComponentModel para o atributo DefaultValue.

2.2.2 Usando o atributo AcceptVerbs

O atributo AcceptVerbs permite definir ações para operações HTTP específicas. Você determina quais operações HTTP, um método de controlador irá responder.

Por exemplo, quando temos um formulário de inserção, usamos dois métodos com o mesmo nome, mas com parâmetros diferentes.

O primeiro é usado para exibir o formulário em branco. É invocado somente por operações HTTP GET:

```
public ActionResult Create() {  
    return View();  
}
```

Ou:

```
[AcceptVerbs(HttpVerbs.Get)]
public ActionResult Create() {
    return View();
}
```

O segundo insere as informações dos campos do formulário no banco de dados, e é invocado somente por operações HTTP POST:

```
[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Create(Product p) {
    if (!ModelState.IsValid) return View();
    db.AddToProducts(p);
    db.SaveChanges();
    return RedirectToAction("Index");
}
```

Exemplo:

```
using System.Web.Mvc;
using AlfredoLotar.Livro.AspnetMvc;
namespace AppCapitulo2.Controllers {
    public class HomeController : Controller {
        NorthwindBD db = new NorthwindBD();
        public ActionResult Index() {
            ViewBag.Mensagem = "ASP.NET MVC!";
            return View();
        }
        public ActionResult Create() {
            return View();
        }
        [AcceptVerbs(HttpVerbs.Post)]
        public ActionResult Create(Product p) {
            try {
                if (!ModelState.IsValid) return View();
                db.AddToProducts(p);
                db.SaveChanges();
                return RedirectToAction("Index");
            }
            finally {
                if (db != null) db.Dispose();
            }
        }
    }
}
```

Obs.: além das operações HTTP GET e POST o atributo `AcceptVerbs` pode ser definido com outros tipos de operações, como: PUT, DELETE, HEAD.

A partir do ASP.NET MVC 2 usamos os atributos [HttpPost], [HttpGet], [HttpPut], [HttpDelete], em vez de [AcceptVerbs(HttpVerbs.Post)], [AcceptVerbs(HttpVerbs.Get)], [AcceptVerbs(HttpVerbs.Put)], [AcceptVerbs(HttpVerbs.Delete)]. Mais fáceis para ler e digitar.

Exemplo:

```
[HttpGet]
public ActionResult Create() {
    return View();
}

[HttpPost]
public ActionResult Create(Product p) {
    if (!ModelState.IsValid) return View();
    try {
        db.AddToProducts(p);
        db.SaveChanges();
        return RedirectToAction("Index");
    }
    finally {
        if (db != null) db.Dispose();
    }
}
```

2.2.3 Atributo ActionName

O atributo ActionName é usado para nomear um método de controlador. No exemplo a seguir, temos dois métodos diferentes que respondem à mesma ação:

```
[ActionName("Create")]
[HttpGet]
public ActionResult CreateGet() {
    return View();
}

[ActionName("Create")]
[HttpPost]
public ActionResult CreatePost(Product p) {
    if (!ModelState.IsValid) return View();
    try {
        db.AddToProducts(p);
        db.SaveChanges();
        return RedirectToAction("Index");
    }
    finally {
        if (db != null) db.Dispose();
    }
}
```

Se preferir, coloque os atributos na mesma linha, separados por vírgula:

```
[HttpPost, ActionName("Create")]
public ActionResult CreatePost(Product p) {...}
```

2.2.4 Atributo NonAction

O atributo `NonAction` indica que um método público de controlador não pode ser invocado ou executado pelo usuário. Desativa o método de controlador. Exemplo:

```
[NonAction]
public ActionResult Details(int? id=5) {
    if (!id.HasValue)
        return RedirectToAction("Index");
    ViewBag.Mensagem = "Detalhes id = " + id;
    return View();
}
```

Ao invocar o método `Details` é retornada uma mensagem de erro, conforme figura 2.2:

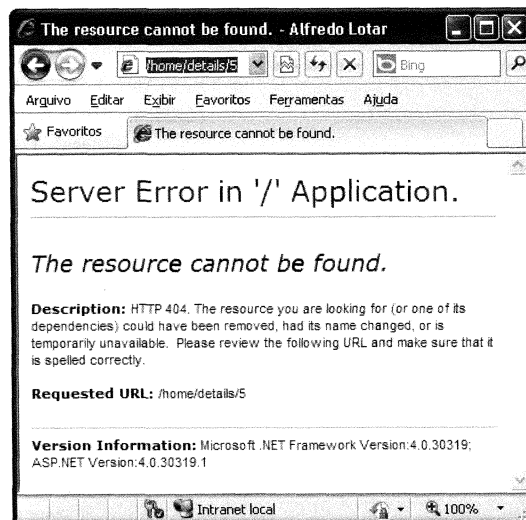


Figura 2.2 – O atributo `NonAction` desativou o método `Details`.

2.2.5 Atributo Bind

Geralmente, usamos o atributo `Bind` para definir as propriedades permitidas.

Exemplo:

```
[HttpPost, ActionName("Create")]
public ActionResult CreatePost([Bind(Exclude = "ProductID, UnitPrice")] Product p) {
    return RedirectToAction("Index");
}
```

A propriedade `Exclude` exclui e a propriedade `Include` define as propriedades usadas pelo método `CreatePost`. Exemplo:

```
[HttpPost,ActionName("Create")]
public ActionResult CreatePost([Bind(Include = "ProductName, CategoryID, QuantityPerUnit,
    Discontinued")] Product p) {
    return RedirectToAction("Index");
}
```

Ou:

```
[HttpPost,ActionName("Create")]
public ActionResult CreatePost([Bind(Include = "ProductName, CategoryID, QuantityPerUnit,
    Discontinued", Exclude = "ProductID")] Product p) {
    return RedirectToAction("Index");
}
```

O atributo `Bind` pode ser aplicado a um parâmetro ou a uma classe. Exemplo:

```
[Bind(Include = "ProductName, CategoryID, QuantityPerUnit, Discontinued")]
public class HomeController : Controller
{}
```

A propriedade `Prefix` associa uma marcação – campo de formulário com um parâmetro de um método de controlador:

```
[HttpPost,ActionName("Create")]
public ActionResult CreatePost([Bind(Prefix = "txt")] Product p)
{}
```

O campo do formulário no view pode ser definido da seguinte forma:

```
<%: Html.TextBox("txt.ProductName")%>
```

2.2.6 Método `HandleUnknownAction`

É possível exibir o conteúdo de um view sem um método de controlador associado? Sim, é possível, basta incluir na classe controladora o seguinte trecho de código:

```
protected override void HandleUnknownAction(string actionName) {
    try {
        this.View(actionName).ExecuteResult(this.ControllerContext);
    }
    catch (InvalidOperationException) {
        this.View("NotFound").ExecuteResult(this.ControllerContext);
    }
}
```

Recurso útil para expor páginas de conteúdo estático como a página “Quem somos” do website.

Por exemplo, quando o usuário digita:

`http://localhost:1034/home/quemsomos`

O view denominado `quemsomos.aspx`, `quemsomos.ascx` ou `quemsomos.cshtml` é invocado e o conteúdo exibido no navegador.

2.2.7 Atributo `HandleError`

Quando ocorre um erro específico na aplicação, o atributo `HandleError` pode ser usado para redirecionar o usuário para um view predeterminado.

Para ativar o atributo `HandleError`, defina o elemento `customErrors` no arquivo de configuração `web.config`:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.web>
    <customErrors mode="On" defaultRedirect="Error" />
  </system.web>
</configuration>
```

E aplique o atributo a uma classe de controlador:

```
[HandleError]
public class HomeController : Controller {}
```

Ou método de controlador:

```
[HandleError]
public ActionResult Index() {
    throw new InvalidOperationException();
    return View();
}
```

Obs.: a palavra-chave `throw` foi usada no exemplo para disparar a exceção automaticamente. Não é obrigatória quando usamos o atributo `HandleError`.

Se necessário, defina as propriedades do atributo `HandleError`:

```
[HandleError(View = "Error", ExceptionType = typeof(InvalidOperationException))]
public ActionResult Index() {
    throw new InvalidOperationException();
    return View();
}
```

A propriedade `View` define o nome do view que deve ser exibido quando o erro ocorrer. E a propriedade `ExceptionType` define o tipo de exceção que o filtro deve capturar.

Obs.: para que o atributo `HandleError` funcione corretamente, é preciso executar a aplicação a partir de um diretório virtual.

2.2.8 Criando atributos

Você pode criar seus próprios atributos com a classe `ActionMethodSelectorAttribute`. O processo é bem simples. Crie uma nova classe derivada de `ActionMethodSelectorAttribute` no diretório `Models` e modifique o método `IsValidForRequest`. Exemplo:

```
using System.Reflection;
using System.Web;
using System.Web.Mvc;
namespace AlfredoLotar.Livro.AspnetMvc {
    public class AuthenticatedAttribute : ActionMethodSelectorAttribute {
        public override bool IsValidForRequest(ControllerContext controllerContext,
            MethodInfo methodInfo) {
            return HttpContext.Current.Request.IsAuthenticated;
        }
    }
}
```

O código personalizado do atributo é colocado no método `IsValidForRequest`:

```
public override bool IsValidForRequest(ControllerContext controllerContext,
    MethodInfo methodInfo) {
    return HttpContext.Current.Request.IsAuthenticated;
}
```

Após compilar a nova classe, a namespace é declarada antes da classe de controlador:

```
using AlfredoLotar.Livro.AspnetMvc;
```

E a classe `AuthenticatedAttribute` é declarada em um método de controlador:

```
[Authenticated]
public ActionResult Admin() {
    return Content("<h1>Usuário autenticado.</h1>");
}
```

Exemplo:

```
using System.Web;
using System.Web.Mvc;
using AlfredoLotar.Livro.AspnetMvc;
namespace AppCapitulo2.Controllers {
    public class HomeController : Controller {
        public ActionResult Index() { return View(); }
        [Authenticated]
        public ActionResult Admin() {
            return Content("<h1>Usuário autenticado.</h1>");
        }
    }
}
```

Somente usuários autenticados conseguem exibir a mensagem “Usuário autenticado.” no navegador.

2.2.9 Atributo SessionState

O estado de sessão do ASP.NET usa variáveis de sessão para armazenar informações únicas e exclusivas de determinado usuário. Por exemplo: itens de um carrinho de compras.

Para ativar, desativar ou tornar somente leitura o estado de sessão do ASP.NET MVC, use a enumeração `SessionStateBehavior` e o atributo `SessionState`. Exemplo:

```
[SessionState(SessionStateBehavior.ReadOnly)]
public class HomeController : Controller {
    public ActionResult Index() {
        // Session["nome"] = "Alfredo Lotar";
        return View();
    }
}
```

Obs.: a enumeração `SessionStateBehavior` pertence à namespace `System.Web.SessionState`.

Para armazenar informações em uma variável de sessão use:

```
Session["nome"] = "Alfredo Lotar";
```

Ou:

```
Session.Add("nome", "Alfredo Lotar");
```

Obs.: quando o estado de sessão do ASP.NET MVC é definido como somente leitura ou é desativado, você não pode armazenar informações em uma variável de sessão.

Para ler o conteúdo de uma variável de sessão, realizamos uma conversão explícita. Exemplo:

```
public ActionResult Ler() {
    if (Session["nome"] != null) {
        return Content(Session["nome"].ToString());
    }
    return RedirectToAction("Index");
}
```

Ou use `<%=Session["nome"]%>` num view.

Alerta: evite ataques de negação de serviço (DOS) utilizando variáveis de sessão somente em páginas restritas, ou seja, protegidas por login.

O identificador de sessão pode ser lido com a propriedade `SessionID`:

```
public ActionResult SessionID() {
    return Content(Session.SessionID);
}
```

Para finalizar uma sessão, use o método `Abandon`.

```
public ActionResult Sair() {  
    Session.Abandon();  
    return RedirectToAction("Index");  
}
```

Obs.: uma sessão é encerrada automaticamente quando o arquivo `Global.asax` ou o arquivo `web.config` é alterado. Reiniciar o servidor no qual a aplicação ASP.NET se encontra também encerra a sessão. Cuidado com softwares antivírus, pois eles podem alterar os arquivos `Global.asax` e `web.config` e assim encerrar uma sessão.

Os eventos `Session_OnStart` e `Session_OnEnd` são manipulados no arquivo `Global.asax`. `Session_OnStart` é executado quando uma nova sessão é iniciada e `Session_OnEnd` quando a sessão é finalizada. Exemplo:

```
protected void Session_Start(object sender, EventArgs e) {  
    Response.Write("Seja bem vindo usuário.");  
}
```

2.2.9.1 Formas de armazenamento

As informações contidas em variáveis de sessão podem ser armazenadas em diferentes meios. Os principais meios de armazenamento suportados são:

Modo	Descrição
InProc	Armazena as informações na memória do servidor, ou seja, no mesmo processo do ASP.NET. É a opção-padrão.
StateServer	Em vez do processo do ASP.NET, usa um processo separado chamado de serviço de estado do ASP.NET para armazenar as informações. Permite o compartilhamento de informações entre servidores.
SQLServer	Armazena as informações em um banco de dados SQL Server. Permite o compartilhamento de informações entre servidores.
Custom	Permite que as informações sejam armazenadas em uma origem de dados personalizada.
Off	Não permite a utilização de variáveis de sessão. Útil em aplicações pequenas.

Exemplo:

```
<?xml version="1.0" encoding="utf-8"?>  
<configuration>  
    <system.web>  
        <sessionState cookieless="false"/>  
    </system.web>  
</configuration>
```

Obs.: as informações contidas nesse tópico foram adaptadas a partir do tópico 4.3 Session state – variáveis de sessão do meu livro Como programar com ASP.NET e C# 2ª edição, também publicado pelo Novatec.

2.2.10 Tipos retornados de ActionResult

Métodos declarados no controlador retornam classes derivadas de `ActionResult`. O tipo retornado depende diretamente da tarefa executada e do método usado. Exemplo, o método `View` retorna `ViewResult` derivado de `ActionResult`.

A seguir listamos as classes derivadas de `ActionResult` e seus respectivos métodos.

Classe derivada de <code>ActionResult</code>	Métodos	Descrição
<code>ViewResult</code>	<code>View</code>	Usado para exibir um view.
<code>PartialViewResult</code>	<code>PartialView</code>	Usado para exibir um user control. Também denominado partial view.
<code>ContentResult</code>	<code>Content</code>	Retorna a saída como texto puro.
<code>EmptyResult</code>		Método de controlador não retorna nada. Semelhante à palavra-chave <code>void</code> .
<code>FileResult</code>	<code>File</code>	A saída é conteúdo de um arquivo.
<code>HttpUnauthorizedResult</code>		Representa acesso não autorizado.
<code>JavaScriptResult</code>	<code>JavaScript</code>	A resposta é enviada como conteúdo JavaScript.
<code>JsonResult</code>	<code>Json</code>	A resposta é enviada em forma de notações JSON.
<code>RedirectResult</code>	<code>Redirect</code> , <code>RedirectPermanent</code>	Redireciona o usuário para uma página específica.
<code>RedirectToRouteResult</code>	<code>RedirectToAction</code> , <code>RedirectToRoute</code> , <code>RedirectToActionPermanent</code> , <code>RedirectToRoutePermanent</code>	Redireciona o usuário para um método de controlador ou rota específica.
<code>HttpNotFoundResult</code>		Retorna uma página não encontrada.
<code>HttpStatusCodeResult</code>	Método definido pelo programador.	Retorna um código de status HTTP definido pelo programador.

2.2.10.1 Retornando ViewResult

A classe `ViewResult` por intermédio do método `View` retorna uma saída HTML. Exemplo:

```
public ActionResult Index() {
    return View();
}
```

O nome do view é implícito, ou seja, o ASP.NET MVC executa o view que tem o mesmo nome do método de controlador. O nome do view executado pode ser definido também explicitamente:

```
public ActionResult Index() {
    return View("Index");
}
```

Ou você pode passar o caminho para o view:

```
public ActionResult Index() {
    return View("~/Views/Home/Index.aspx");
}
```

O model usado também pode ser passado para o método View:

```
private NorthwindBD db = new NorthwindBD();
public ActionResult Index() {
    try {
        var model = db.Categories.AsParallel().ToList();
        return View("Index", model);
    }
    finally {
        if (db != null) db.Dispose();
    }
}
```

Assim como a master page:

```
private NorthwindBD db = new NorthwindBD();
public ActionResult Index() {
    try {
        var model = db.Categories.AsParallel().ToList();
        return View("Index", "ViewMasterPage1");
    }
    finally {
        if (db != null) db.Dispose();
    }
}
```

Ou ambos:

```
private NorthwindBD db = new NorthwindBD();
public ActionResult Index() {
    try {
        var model = db.Categories.AsParallel().ToList();
        return View("Index", "ViewMasterPage1", model);
    }
    finally {
        if (db != null) db.Dispose();
    }
}
```

2.2.10.2 Retornando PartialViewResult

Pode ser usado para carregar trechos de uma página contidos em um user control. Por exemplo, o user control – partial view ViewMenu.ascx contém um menu simples:

```
<ul>
    <li>Home</li>
    <li>Eletrônica</li>
    <li>Idiomas</li>
    <li>Negócios</li>
```

```

    <li>Programação</li>
    <li>Segurança</li>
    <li>UML</li>
</ul>

```

Exibido com o método PartialView:

```

public ActionResult Menu() {
    return PartialView("ViewMenu");
}

```

Exemplo:

```

using System;
using System.Web.Mvc;
namespace AppCapitulo2.Controllers {
    public class HomeController : Controller {
        public ActionResult Index() {
            return View();
        }
        public ActionResult Menu() {
            return PartialView("ViewMenu");
        }
    }
}

```

Restrinja o acesso direto ao user control com o atributo ChildActionOnly:

```

[ChildActionOnly]
public ActionResult Menu() {
    return PartialView("ViewMenu");
}

```

Para usar o user control ViewMenu.ascx em uma página qualquer, basta escrever uma única linha de código:

```

<%:Html.Action("Menu") %>

```

Exemplo:

```

<!DOCTYPE html>
<html>
    <head>
        <title>Index</title>
    </head>
    <body>
        <div>
            <%:Html.Action("Menu") %>
        </div>
    </body>
</html>

```

Observe a figura 2.3:

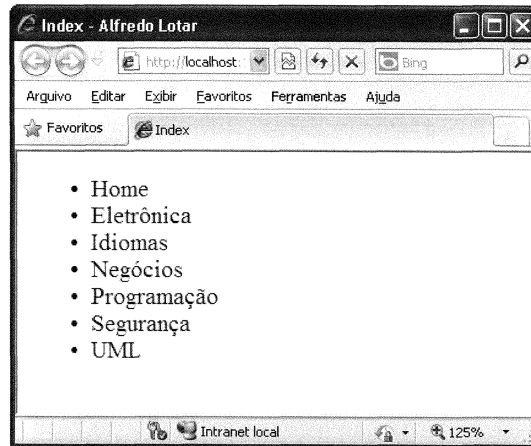


Figura 2.3 – Menu carregado a partir de um user control.

2.2.10.3 Retornando ContentResult

Retorna qualquer tipo de texto. Há duas formas de processar texto. A primeira usa um método de controlador que retorna uma string:

```
public string Index() {  
    return "Isto é um texto simples";  
}
```

A segunda forma usa o método Content:

```
public ActionResult Index() {  
    return Content("Isto é um texto simples");  
}
```

O segundo parâmetro do método Content usa codificação Multipurpose Internet Mail Extensions (MIME):

```
public ActionResult Index() {  
    return Content("Isto é um texto simples", "text/html");  
}
```

A lista completa de tipos MIME você encontra no link a seguir:

<http://www.iana.org/assignments/media-types/index.html>

O terceiro parâmetro determina a codificação dos caracteres. Útil para textos em português, com acentos.

Exemplo:

```
public ActionResult Index() {
    return Content("Isto é um texto simples", "text/html", Encoding.UTF8);
}
```

Ou:

```
public ActionResult Index() {
    return Content("Isto é um texto simples", "text/html", Encoding.GetEncoding("iso-8859-1"));
}
```

2.2.10.4 Retornando FileResult

Para abrir um arquivo direto no navegador, use:

```
public ActionResult Index() {
    return File("aspnetmvc3.pdf", "application/pdf");
}
```

Observe a figura 2.4:

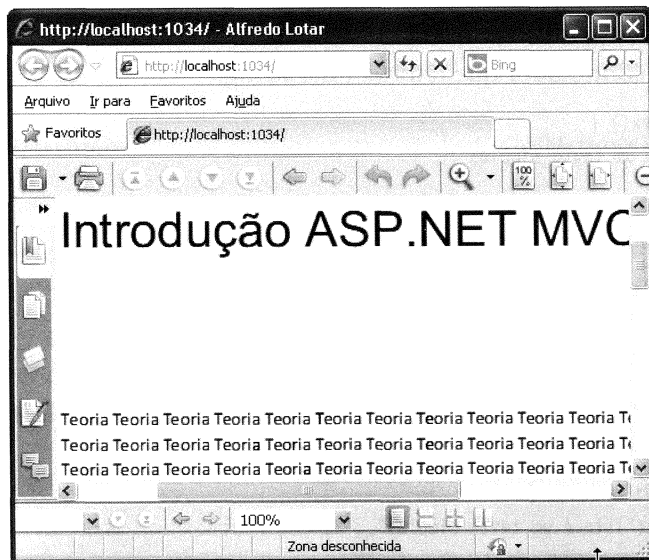


Figura 2.4 – Arquivo .PDF no navegador.

Ou permita o download de arquivos:

```
public ActionResult Index() {
    return File("aspnetmvc3.pdf", "application/pdf", "Salvarcomo.pdf");
}
```

Observe a figura a seguir:

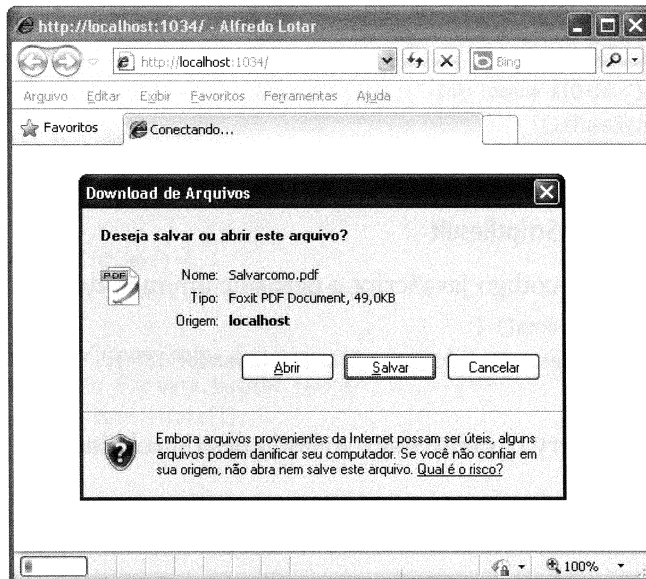


Figura 2.5 – Download de arquivos.

Ou carregue websites:

```
public ActionResult Website() {
    WebClient cliente = new WebClient();
    Stream stream = cliente.OpenRead("http://www.novatec.com.br/");
    return File(stream, "text/html");
}
```

Figuras armazenadas em um banco de dados SQL Server:

```
public ActionResult Website() {
    byte[] b = GetLogo();
    return File(b, "image/gif");
}
```

O código do método `GetLogo` extrai a imagem do banco de dados SQL Server. O código deve ser semelhante ao do artigo “Retrieving Large Data (ADO.NET)”, localizado no website da Microsoft: <http://msdn.microsoft.com/en-us/library/87z0hy49.aspx>

2.2.10.5 Retornando `EmptyResult`

Ocorre quando um método de controlador não retorna valor.

Exemplo:

```
public void Index() {
    Response.Write("<h1>Olá mundo!</h1>");
}
```

É o mesmo que:

```
public ActionResult Index() {
    Response.Write("<h1>Olá mundo!</h1>");
    return new EmptyResult();
}
```

2.2.10.6 Retornando JavaScriptResult

Permite a execução de código JavaScript a partir de um método de controlador:

```
public JavaScriptResult Alerta() {
    return new JavaScriptResult() { Script = "alert('Olá mundo!');" };
}
```

Em um view use JQuery para invocar o método de controlador:

```
@{ Layout = null; }
<!DOCTYPE html>
<html>
    <head>
        <title>Retornando JavaScriptResult</title>
        <script src="@Url.Content("~/Scripts/jquery-1.4.4.js")" type="text/javascript"></script>
        <script type="text/javascript">
            $.getScript("/Home/alerta");
        </script>
    </head>
    <body>
        <div>
        </div>
    </body>
</html>
```

2.2.10.7 Retornando JsonResult

Retorna uma notação JSON:

```
public ActionResult List() {
    var model = new List<Cidades>() {
        new Cidades { CidadeID = 1, Nome = "Pelotas", Estado = "RS" },
        new Cidades { CidadeID = 2, Nome = "Curitiba", Estado = "PR" },
        new Cidades { CidadeID = 3, Nome = "São Paulo", Estado = "SP" },
        new Cidades { CidadeID = 4, Nome = "Rio de Janeiro", Estado = "RJ" },
        new Cidades { CidadeID = 5, Nome = "Florianópolis", Estado = "SC" },
        new Cidades { CidadeID = 6, Nome = "Salvador", Estado = "BA" }
    };
    return Json(model, JsonRequestBehavior.AllowGet);
}
```

No view usamos JQuery para processar a saída. Crie um view chamado `Lista.cshtml` no diretório `Views/Home`.

```

@{ Layout = null; }
<!DOCTYPE html>
<html>
  <head>
    <title>Lista de Cidades</title>
    <script src="../../Scripts/jquery-1.4.4.js" type="text/javascript"></script>
    <script type="text/javascript">
      $(getCidades);
      function getCidades() {
        $.getJSON("/home/List", showCidades);
      }
      function showCidades(data) {
        for (i = 0; i < data.length; i++) {
          var cidade = data[i];
          $("#IdCidades").append("<li>" + cidade.Nome + "</li>");
        }
      }
    </script>
  </head>
  <body>
    <div>
      <b>Lista de Cidades:</b>
      <ul>
        <div id="IdCidades">
        </ul>
      </div>
    </body>
  </html>

```

No Visual Studio pressione F5 e acesse: <http://localhost:NúmeroDaPorta/home/lista>

Exemplo: <http://localhost:1226/home/lista>

O resultado vemos na figura 2.6:

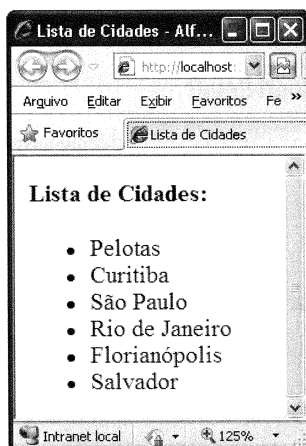


Figura 2.6 – Lista de cidades exibida com JSON.

Certifique-se que a classe controladora possui o método `HandleUnknownAction`:

```
protected override void HandleUnknownAction(string actionName) {
    try {
        this.View(actionName).ExecuteResult(this.ControllerContext);
    }
    catch (InvalidOperationException) {
        this.View("NotFound").ExecuteResult(this.ControllerContext);
    }
}
```

E o view `NotFound`:

```
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
    <head>
        <title>NotFound</title>
    </head>
    <body>
        <div>
            NotFound
        </div>
    </body>
</html>
```

2.2.10.8 Retornando `RedirectResult`

O método `Redirect` retorna `RedirectResult` e redireciona o usuário para um website específico:

```
public ActionResult Index() {
    return Redirect("http://www.novatec.com.br");
}
```

Você pode usar também o método `RedirectPermanent`.

2.2.10.9 Retornando `RedirectToRouteResult`

Para redirecionar o usuário para outro método de controlador, use `RedirectToAction`:

```
public ActionResult Index() {
    return RedirectToAction("Edit", "Home");
}
```

`Edit` é o método de controlador e `Home` o prefixo do controlador.

Se preferir, use o nome da rota:

```
public ActionResult Details() {
    return RedirectToRoute("NomeDaRota");
}
```

O nome da rota é definido no arquivo `Global.asax`:

```
public static void RegisterRoutes(RouteCollection routes) {
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
    routes.MapRoute(
        "NomeDaRota",
        "{controller}/{action}/{id}",
        new { controller = "Home", action = "Index", id = UrlParameter.Optional } );
}
```

2.2.10.10 Retornando `HttpNotFoundResult`

Quando o usuário acessa um recurso inexistente, acione o método `HttpNotFound` e redirecione-o para uma página de erro específica:

```
public ActionResult Details(int? id) {
    if (!id.HasValue)
        return HttpNotFound();
    ViewBag.Mensagem = "Detalhes id = " + id;
    return View();
}
```

Ou:

```
public ActionResult Details(int? id) {
    if (!id.HasValue)
        return new HttpNotFoundResult();
    ViewBag.Mensagem = "Detalhes id = " + id;
    return View();
}
```

2.2.10.11 Retornando `HttpStatusCodeResult`

Se preferir, use o código da mensagem com a classe `HttpStatusCodeResult`:

```
public ActionResult Details(int? id) {
    if (!id.HasValue)
        return new HttpStatusCodeResult(404);
    ViewBag.Mensagem = "Detalhes id = " + id;
    return View();
}
```

2.2.10.12 Retornando `HttpUnauthorizedResult`

Acione a classe `HttpUnauthorizedResult` quando o usuário acessa um recurso protegido ao qual não tem acesso. Exemplo:

```
public ActionResult Login() {
    try {
        Response.Write(Mensagem("Você está acessando uma página do grupo 'Admin'."));
    }
}
```

```

        catch (SecurityException) {
            return new HttpUnauthorizedResult();
        }
        return View();
    }

    [PrincipalPermission(SecurityAction.Demand, Role = "Administradores")]
    private string Mensagem(string str) {
        return str;
    }
}

```

Quando o código aciona a classe `HttpUnauthorizedResult`, o usuário é redirecionado para a página de login definida no elemento `forms` do arquivo `web.config`:

```

<authentication mode="Forms">
    <forms loginUrl="~/Account/LogOn" timeout="2880" />
</authentication>

```

2.3 Cache

É usado para aumentar a performance do website. A saída do cache pode estar no cliente, no servidor web, em um servidor proxy. O cache reduz o tráfego da rede, pois a página não precisa acessar o banco de dados constantemente.

Coloque no cache páginas muito acessadas e com conteúdo estático. Não inclua no cache páginas atualizadas frequentemente.

Obs.: cuidado para não colocar muitas páginas no cache, pois isso diminui a memória disponível no servidor, podendo até prejudicar a performance da aplicação web.

No ASP.NET MVC o cache é ativado com o atributo `OutputCache`:

```

[OutputCache(Duration = 60, VaryByParam = "none")]
public ActionResult List() {
    return View();
}

```

O qual possui inúmeros parâmetros, conforme lista a seguir:

Parâmetro	Descrição
Duration	Determina o tempo em que a página permanece no cache. O valor é em segundos.
Location	Define o local em que o cache é armazenado. Exemplo, na memória do servidor, no navegador do cliente, servidor proxy.
VaryByParam	Armazena múltiplas versões de uma página no cache baseando-se nos valores da query string e/ou campos de um formulário.
VaryByHeader	São armazenadas versões diferentes da página baseando-se nos valores dos cabeçalhos de resposta.

Parâmetro	Descrição (cont.)
VaryByCustom	São armazenadas versões diferentes da página com base em parâmetros personalizados.
VaryByContentEncoding	Coloca no cache páginas diferentes de acordo com valores do cabeçalho Accept-Encoding.
NoStore	Se for definido como <code>true</code> , as informações contidas no cache não são armazenadas em um servidor proxy ou navegador.
CacheProfile	Instrui o ASP.NET MVC a acessar as informações sobre o cache em um arquivo <code>web.config</code> .
SqlDependency	Define informações referentes às dependências do cache em relação ao SQL Server.

2.3.1 Parâmetro Duration

O parâmetro `Duration` define o tempo em que a página permanece no cache. O valor é em segundos. O tempo do cache é importantíssimo. Em um website com muitos acessos o tempo de duração do cache deve ser de poucos segundos, enquanto websites com poucos visitantes devem avaliar a real necessidade de cache.

Lembrando que o atributo `OutputCache` pode ser definido para todos os métodos do controlador:

```
using System.Web.Mvc;
namespace AppCapitulo2.Controllers {
    [OutputCache(Duration = 60, VaryByParam = "none")]
    public class HomeController : Controller {
        public ActionResult Index() {
            return View();
        }
    }
}
```

Ou somente para um ou mais métodos de controlador:

```
[OutputCache(Duration = 60, VaryByParam = "none")]
public ActionResult Index() {
    return View();
}
```

2.3.2 Cache com múltiplas versões de uma página

Uma determinada página ASP.NET pode ter diferentes versões dependendo do tipo de parâmetro passado para a página. A saída do cache pode variar com base:

- Em query string.
- Campos de formulário.
- Cabeçalhos de resposta.

- Versão do navegador.
- Parâmetros personalizados.

O atributo `OutputCache` utiliza os seguintes parâmetros para criar múltiplas versões de uma página: `VaryByParam`, `VaryByHeader`, `VaryByCustom`, `VaryByContentEncoding`.

2.3.2.1 Parâmetro `VaryByParam`

O parâmetro `VaryByParam` pode ser definido como: `None`, `*`, com uma lista de query strings e campos de formulário (POST). Observe a tabela a seguir.

Valores	Descrição
<code>None</code>	Realiza o cache de uma única página.
<code>Request.QueryString</code>	Armazena múltiplas versões de uma página no cache baseando-se nos valores de uma query string.
<code>Request.Form</code>	Armazena múltiplas versões de uma página no cache baseando-se nos valores de campos de um formulário.
<code>*</code>	Armazena múltiplas versões de uma página no cache baseando-se nos valores da query string e/ou campos de um formulário.

A página a seguir exibe nomes cujo prefixo é M:

`http://www.siteexemplo.com/resultado.aspx?letra=M`

Nesse caso, é criada uma página diferente para cada letra do alfabeto.

Para colocar no cache todas as versões da página, utilize:

```
[OutputCache(Duration = 60, VaryByParam = "letra")]
public ActionResult List() {
    return View();
}
```

Isso mantém cada versão da página por 60 segundos no cache.

Obs.: os nomes dos parâmetros são sensíveis a letras maiúsculas e minúsculas. Se você criar uma página com um parâmetro igual à `letra` e outra com o parâmetro `Letra`, o ASP.NET adiciona duas versões da mesma página ao cache.

Se a página tiver uma lista de parâmetros, separe os parâmetros com ponto e vírgula (;):

```
[OutputCache(Duration = 60, VaryByParam = "letra;curso")]
public ActionResult List() {
    return View();
}
```

2.3.2.2 Parâmetro `VaryByHeader`

O ASP.NET põe no cache múltiplas versões de uma página com base em valores do cabeçalho HTTP:


```
[OutputCache(Duration = 60, VaryByParam = "none", VaryByHeader = "Accept-Language")]
public ActionResult List() {
    return View();
}
```

No exemplo, as versões das páginas são divididas de acordo com o idioma configurado no computador do visitante do website. Recomendável a websites que recebem muitos visitantes de um único país.

Obs.: o parâmetro `VaryByHeader` segue as mesmas regras do parâmetro `VaryByParam` para a utilização de múltiplos parâmetros.

2.3.2.3 Parâmetro `VaryByContentEncoding`

Coloca no cache páginas diferentes de acordo com valores do cabeçalho `Accept-Encoding`. Geralmente é definido como `gzip` ou `deflate`. Exemplo:

```
[OutputCache(Duration = 20, VaryByParam = "none", VaryByContentEncoding = "deflate")]
public ActionResult List() {
    return View();
}
```

2.3.2.4 Parâmetros personalizados

Além dos cabeçalhos HTTP e das query strings, podemos enviar múltiplas versões de uma página para o cache com base em parâmetros personalizados desenvolvidos por nós, programadores. No método de controlador, adicione a seguinte declaração:

```
[OutputCache(Duration = 20, VaryByParam = "none", VaryByCustom = "navegador")]
```

Exemplo:

```
[OutputCache(Duration = 20, VaryByParam = "none", VaryByCustom = "navegador")]
public ActionResult List() {
    return View();
}
```

Em seguida, adicione o método `GetVaryByCustomString` ao arquivo `Global.asax`:

```
public override string GetVaryByCustomString(System.Web.HttpContext context, string arg) {
    if (arg.Equals("navegador")) {
        return context.Request.Browser.ToString();
    }
    else {
        return "";
    }
}
```

Pronto, a página ASP.NET envia para o cache múltiplas versões usando o parâmetro `navegador`.

2.3.2.5 Parâmetro Location

Quando o servidor web envia a resposta para uma requisição do navegador, o servidor inclui na resposta o campo `Cache-Control` no cabeçalho HTTP. Esse campo define o local em que a saída do cache pode ser armazenada.

Os valores do campo `Cache-Control` podem ser definidos com o parâmetro `Location`.

O parâmetro `Location` possui os seguintes valores:

Valor	Descrição
Any	Esse é o valor-padrão. A saída do cache pode estar localizada, em qualquer lugar. Exemplo: no cliente, no servidor proxy ou em qualquer outro servidor que faça parte da requisição.
Client	A saída do cache está localizada somente no navegador do cliente.
Downstream	A saída do cache está localizada em qualquer servidor proxy ou navegador.
Server	A saída do cache está localizada no servidor web.
ServerAndClient	A saída do cache está localizada no cliente ou no servidor.
None	A saída do cache está desativada.

Exemplo:

```
[OutputCache(Duration = 60, Location = OutputCacheLocation.Client, VaryByParam = "none")]
public ActionResult List() {
    return View();
}
```

2.3.2.6 Parâmetro NoStore

Quando definimos o parâmetro `NoStore` como `true`, as informações contidas no cache não são armazenadas em um servidor proxy ou navegador. Útil quando temos informações sigilosas no cache.

```
[OutputCache(Duration = 20, VaryByParam = "none", NoStore = true)]
public ActionResult List () {
    return View();
}
```

Obs.: por motivos de segurança, evite armazenar informações sigilosas no cache.

2.3.2.7 Parâmetro CacheProfile

O parâmetro `CacheProfile` permite que você use uma única declaração de cache em inúmeras páginas. Por exemplo, você declara as informações de cache no elemento `caching` do arquivo `web.config`:

```
<configuration>
  <system.web>
    <caching>
      <outputCacheSettings>
        <outputCacheProfiles>
```

```

        <add name="CacheDoLivro" duration="60" varyByParam="none"/>
    </outputCacheProfiles>
</outputCacheSettings>
</caching>
</system.web>
</configuration>

```

Após a definição das informações no arquivo `web.config`, aplique o atributo `OutputCache` e o parâmetro `CacheProfile` ao método ou classe de controlador:

```

[OutputCache(CacheProfile = "CacheDoLivro")]
public ActionResult List() {
    return View();
}

```

2.3.2.8 Parâmetro `SqlDependency`

O ASP.NET MVC usa a classe `SqlCacheDependency` para monitorar as alterações em tabelas ou linhas de uma tabela de um banco de dados SQL Server. Quando ocorre alguma alteração no banco de dados, o item do cache criado é invalidado e removido.

Você pode tornar um item dependente de uma tabela no SQL Server 7.0, no SQL Server 2000 e no SQL Server 2005 e 2008. Se você usa estes dois últimos, pode também definir a dependência de uma linha específica.

Para ativar o mecanismo de cache dependente do SQL Server, configure a string de conexão no arquivo `web.config`:

```

<connectionStrings>
    <add name="nwind" connectionString="Data Source=.\SQLEXPRESS;Integrated Security=SSPI;
        Initial Catalog=northwind;"/>
</connectionStrings>

```

E, em seguida, o elemento `sqlCacheDependency`:

```

<caching>
    <sqlCacheDependency enabled="true" pollTime="30000">
        <databases>
            <add name="northwind" connectionStringName="nwind" />
        </databases>
    </sqlCacheDependency>
</caching>

```

Exemplo:

```

<?xml version="1.0" encoding="utf-8"?>
<configuration>
    <connectionStrings>
        <add name="nwind" connectionString="Data Source=.\SQLEXPRESS;Integrated Security=SSPI;
            Initial Catalog=northwind;"/>
    </connectionStrings>

```

```

</connectionStrings>
<system.web>
  <compilation debug="false" targetFramework="4.0"/>
  <caching>
    <sqlCacheDependency enabled="true" pollTime="30000">
      <databases>
        <add name="northwind" connectionStringName="nwind" />
      </databases>
    </sqlCacheDependency>
  </caching>
</system.web>
</configuration>

```

Habilite o cache para uma tabela específica:

```
[OutputCache(Duration = 60, VaryByParam = "none", SqlDependency = "BancoDeDados:Tabela")]
```

Exemplo:

```

[OutputCache(Duration = 60, VaryByParam = "none", SqlDependency = "northwind:Products")]
public ActionResult List() {
    return View();
}

```

2.3.3 Cache parcial

Quando não há necessidade de enviar para o cache a página inteira, fragmentamos partes da página e a incluímos em um user control, o qual possui o conteúdo enviado para o cache. Para colocar um user control no cache, inclua a seguinte declaração no topo do método de controlador que contém o código do user control:

```

[ChildActionOnly]
[OutputCache(Duration = 60, VaryByCustom = "none")]
public ActionResult Menu() {
    return PartialView("ViewMenu");
}

```

2.3.4 Adicionando itens para o cache

Adicione itens ao cache com o método `Insert`. Esse método tem diversos parâmetros que nos permitem definir dependências, o tempo de duração do cache, políticas de prioridade, além de uma chave e de um valor.

Podemos adicionar um item ao cache de forma direta:

```
HttpContext.Cache["CacheItem1"] = "valor do cache";
```

Ou com o método `Insert`:

```
HttpContext.Cache.Insert("CacheItem1", "valor do cache");
```

Exemplo:

```
private NorthwindBD db = new NorthwindBD();
public ActionResult Index() {
    System.Collections.Generic.List<Category> model = null;
    if (HttpContext.Cache["CacheItem1"] == null) {
        try {
            model = db.Categories.AsParallel().ToList();
            HttpContext.Cache.Insert("CacheItem1", model, null, DateTime.Now.AddSeconds(60),
                System.Web.Caching.Cache.NoSlidingExpiration);
        }
        finally {
            if (db != null) db.Dispose();
        }
    }
    else {
        model = HttpContext.Cache["CacheItem1"] as System.Collections.Generic.List<Category>;
    }
    return View(model);
}
```

Obs.: cuidado para não criar dois itens com o mesmo nome, pois isso anula o item anteriormente criado.

2.3.5 Extrair itens do cache

Antes de extrair um item do cache, é preciso verificar se esse item existe, pois os dados já podem ter sido excluídos devido a políticas de prioridade ou em razão de o cache já ter expirado:

```
string cacheSaida;
if (HttpContext.Cache["CacheItem1"] != null) {
    cacheSaida = HttpContext.Cache["CacheItem1"] as string;
}
else {
    HttpContext.Cache.Insert("CacheItem1", "valor do cache");
    cacheSaida = HttpContext.Cache["CacheItem1"] as string;
}
Response.Write(cacheSaida);
```

2.3.6 Excluindo itens do cache

Os dados no cache são voláteis, ou seja, não ficam permanentemente armazenados.

Um item é excluído automaticamente do cache quando:

- O servidor está com pouca memória.
- O item do cache expirou.
- O item foi excluído em virtude de mudanças em algum recurso dependente.
- Servidor web é reiniciado.

Para excluir um item do cache de forma explícita, utilize o método `Remove`:

```
HttpContext.Cache.Remove("CacheItem1");
```

2.4 Controladores assíncronos

As operações de entrada e saída são as principais candidatas a serem processadas de forma assíncrona. Operação assíncrona pode ser a leitura de arquivos grandes, acesso a recursos localizados em uma rede, acesso a web services, leitura de dados de um banco de dados, ou de portas COM. Conexões assíncronas não aumentam a performance da aplicação, mas permitem a execução de tarefas paralelas, enquanto você está carregando uma imagem ou vídeo de vários mega bytes, por exemplo.

Um método de controlador assíncrono é derivado de `AsyncController`:

```
public class HomeController : AsyncController {
    // Código aqui
}
```

Uma operação assíncrona envolve dois métodos. Um começa a requisição e o nome contém o sufixo `Async`. Outro completa a requisição e o nome contém o sufixo `Completed`:

```
public class HomeController : AsyncController {
    public void OpenPageAsync() {
        // Código aqui
    }
    public ActionResult OpenPageCompleted(string pageHtml) {
        // Código aqui
    }
}
```

Exemplo:

```
using System;
using System.Net;
using System.Web.Mvc;
namespace AppCapitulo2.Controllers {
    public class HomeController : AsyncController {
        public ActionResult Index() {
            return View();
        }
        public void OpenPageAsync() {
            AsyncManager.OutstandingOperations.Increment();
            var client = new WebClient();
            client.DownloadStringCompleted += (sender, e) =>{
                AsyncManager.Parameters["pageHtml"] = e.Result;
                AsyncManager.OutstandingOperations.Decrement();
            };
        }
    }
}
```

```

        client.DownloadStringAsync(new Uri("http://www.novatec.com.br"));
    }
    public ContentResult OpenPageCompleted(string pageHtml) {
        return Content(pageHtml);
    }
}

```

É necessário informar ao ASP.NET MVC quantas operações estão pendentes. A propriedade `OutstandingOperations` gerencia essa tarefa, enquanto a propriedade `Parameters` define os parâmetros passados para o método `OpenPageCompleted`.

Invocar um método de controlador assíncrono é igual a invocar um método síncrono:

`http://localhost:1034/home/openpage`

Ou:

`http://localhost/dirvirtual/home/openpage`

Você invoca o prefixo do método sem o sufixo `Async`.

2.4.1 Atributos com métodos assíncronos

Atributos como `Authorize`, `OutputCache`, `ActionName`, `AcceptVerbs` etc. devem ser aplicados ao método de controlador com o sufixo `Async`. Exemplo:

```

[Authorize]
public void OpenPageAsync() {
    // Código aqui
}
public ActionResult OpenPageCompleted(string pageHtml) {
    // Código aqui
}

```

O atributo `AsyncTimeout` define o intervalo de tempo que o método assíncrono é executado. O valor-padrão é de 45 segundos. Exemplo:

```

[AsyncTimeout(60)]
public void OpenPageAsync() {
    // Código aqui
}
public ActionResult OpenPageCompleted(string pageHtml) {
    // Código aqui
}

```

Enquanto o atributo `NoAsyncTimeout` define o intervalo de tempo como infinito:

```

[NoAsyncTimeout]
public void OpenPageAsync() {

```

```

    // Código aqui
}
public ActionResult OpenPageCompleted(string pageHtml) {
    // Código aqui
}

```

2.4.2 Outras operações assíncronas

Algumas classes do .NET Framework executam operações assíncronas por intermédio de métodos pares com prefixo `Begin` e `End`, como: `BeginRead`, `EndRead`, `BeginWrite`, `EndWrite`, `BeginGetResponse`, `EndGetResponse`, `BeginExecuteReader`, `EndExecuteReader`, `BeginExecuteNonQuery`, `EndExecuteNonQuery`, `BeginExecuteXmlReader` e `EndExecuteXmlReader` etc. Controladores assíncronos podem realizar operações com esses métodos também.

Exemplo:

```

SqlConnection conn = null;
public void LerAsync() {
    AsyncManager.OutstandingOperations.Increment();
    ConnectionStringSettings getString = WebConfigurationManager.ConnectionStrings["nwind"] as
        ConnectionStringSettings;
    if (getString != null) {
        string sSql = "Select ProductName from Products";
        conn = new SqlConnection(getString.ConnectionString);
        SqlCommand command = new SqlCommand(sSql, conn);

        conn.Open();
        command.BeginExecuteReader(asyncResult =>{
            using (SqlDataReader r = command.EndExecuteReader(asyncResult)) {
                var lst = new List<string>();
                int produto = r.GetOrdinal("ProductName");
                if (r.HasRows)
                    while (r.Read()) {
                        lst.Add(r.GetString(produto).ToString() + "<br/>");
                        AsyncManager.Parameters["reader"] = lst;
                    }
                AsyncManager.OutstandingOperations.Decrement();
            }
        }, null);
    }
}

public ActionResult LerCompleted(IEnumerable<string> reader) {
    if (conn != null) conn.Close();
    return Content(reader.First());
}

```


View

Um view contém o conteúdo e o design da página de um website. Na verdade, contém o texto da página, o código HTML, as imagens, os links, as referências para arquivos de linguagens de script e folhas de estilo CSS, entre outros.

Os mecanismos de exibição como o Razor definem a sintaxe usada na página do view.

Falando em páginas, elas podem conter diferentes extensões, por exemplo: `.aspx`, `.cshtml`, `.master` e `.ascx` – no caso dos user controls, também chamados de partial view.

O modo mais fácil de se criar um view rapidamente é clicando em um método de controlador com o botão direito do mouse e, em seguida, em **Add View...**. Observe a figura 1.15.

Após a criação do view, o designer deve formatar a página manualmente, inserindo layout, texto, imagens etc.

Podemos afirmar que o ASP.NET MVC facilita a vida do programador, mas não a do designer, pois todos os elementos visuais precisam ser criados manualmente.

3.1 Desenvolvendo uma nova aplicação

Antes de começarmos a abordar com mais profundidade os view do ASP.NET MVC, precisamos preparar o terreno, ou seja, inicie o Visual Studio. Acesse o menu **File** e clique em **New Project...**

Na caixa de diálogo **New Project**, selecione **Web > Visual Studio 2012 > ASP.NET MVC 4 Web Application**. Nomeie o projeto como `AppCapitulo3`. Clique em **OK**.

Copie o arquivo `Content/Site.css` e a master page `Views/Shared/Site.Master` da aplicação `AppCapitulo1` para a aplicação `AppCapitulo3`.

Em seguida, acrescente a classe da camada de dados que será usada pelos exemplos deste capítulo. Clique com o botão direito do mouse no diretório `Models` e, em seguida, acrescente uma nova classe. Nomeie-a como `Países.cs`.

Modifique a classe recém-criada substituindo-a pelo código a seguir:

```

using System;
namespace AppCapitulo3.Models {
    public class Paises {
        public string Nome { get; set; }
        public string Continente { get; set; }
        public string Idioma { get; set; }
    }
}

```

Em seguida, compile a aplicação conforme a figura 3.1:

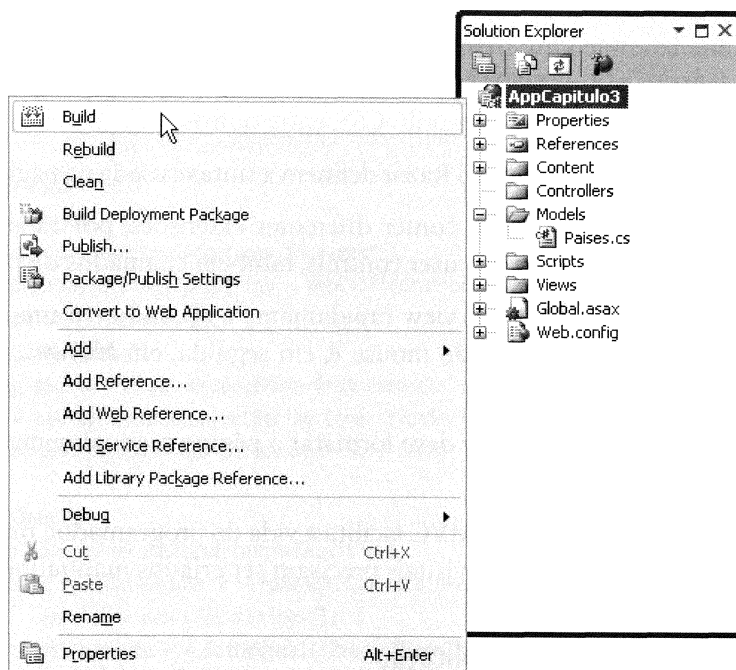


Figura 3.1 – Compilando a aplicação ASP.NET MVC.

Acrescente a aplicação, o controlador HomeController e altere o método Index com o código a seguir:

```

public ActionResult Index() {
    ViewBag.Message = "ASP.NET MVC!";
    ViewData["Mensagem"] = "Países";
    TempData["OutraMensagem"] = "Teste";

    var lst = new List<Paises> {
        new Paises { Nome="Brasil", Continente="América do sul", Idioma="Português" },
        new Paises { Nome="Alemanha", Continente="Europa", Idioma="Alemão" },
        new Paises { Nome="Austrália", Continente="Oceania", Idioma="Inglês" },
        new Paises { Nome="Uruguai", Continente="América do sul", Idioma="Espanhol" },
    }
}

```

```

        new Países { Nome="Japão", Continente="Ásia", Idioma="Japonês" }
    };
    return View(lst);
}

```

Adicione também uma referência para a namespace `AppCapítulo3.Models`:

```
using AppCapítulo3.Models;
```

3.2 Adicionando um view

Clique com o botão direito do mouse no método `Index`. Em seguida, selecione **Add View...**. Na caixa de diálogo **Add View** configuramos o view que está sendo criado. Na caixa de combinação **Model class** selecione a classe `Países`. Observe a figura 3.2:

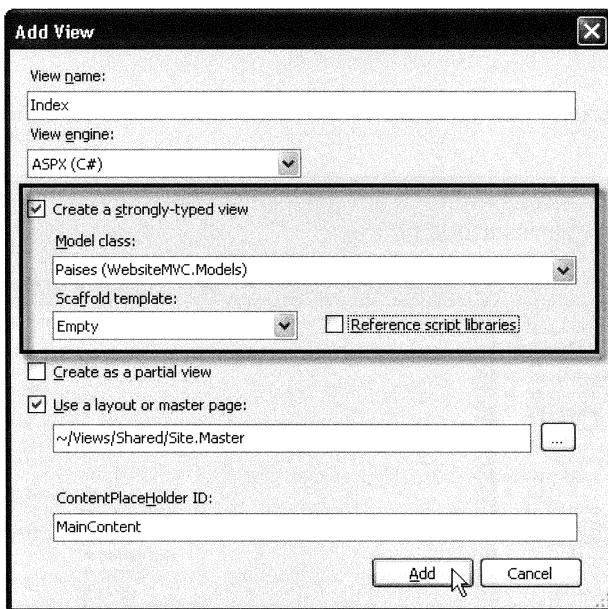


Figura 3.2 – Caixa de diálogo Add View.

O nome do view é preenchido automaticamente, assim como o nome da master page. Na caixa de diálogo **Add View** podemos configurar as seguintes opções:

Opção	Descrição
View name	Nome do view.
View engine	Mecanismo de exibição usado pelo view. Pode ser ASPX ou Razor.
Create a strongly-typed view	Associa um tipo ao view.
Model class	Classe associada ao view.
Scaffold template	Permite definir um modelo predefinido para o view. Pode ser definido como: Create, Delete, Details, Edit, Empty e List.

Opção	Descrição (cont.)
Reference script libraries	Acrescenta ao view referência para arquivos de script. Usado nos templates Create e Edit para fins de validação.
Create as a partial view	Define o view como um user control. Arquivo .ascx.
Use a layout or master page	Contém o caminho e o nome da master page.
ContentPlaceHolder ID	Contém o ID do ContentPlaceHolder definido na master page.

Após clicar no botão **Add** é gerado um novo view. O código gerado depende das opções selecionadas na caixa de diálogo Add View. Por exemplo, quando selecionamos uma master page para o view, temos o código a seguir:

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
    Inherits="System.Web.Mvc.ViewPage<AppCapitulo3.Models.Paises>" %>
<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
    Index
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
    <h2>Index</h2>
</asp:Content>
```

Sem uma master page associada temos:

```
<%@ Page Language="C#" Inherits="System.Web.Mvc.ViewPage<AppCapitulo3.Models.Paises>" %>
<!DOCTYPE html>
<html>
    <head runat="server">
        <title>Index</title>
    </head>
    <body>
        <div>
        </div>
    </body>
</html>
```

Obs.: a opção Scaffold template igual a Empty gera uma página sem conteúdo.

Para executar expressões dentro de um view e retornar o resultado, use <%= %>.

Exemplo:

```
<%= ViewBag.Message %>
```

Você pode usar também <%= %> para retornar o valor de ViewData:

```
<%= ViewData["Mensagem"] %>
```

E de TempData:

```
<%= TempData["OutraMensagem"] %>
```

TempData difere um pouco dos demais (ViewBag e ViewData), pois os dados permanecem armazenados somente de uma requisição para outra.

Obs.: ViewData era muito utilizado nas versões 1 e 2 do ASP.NET MVC. A partir da versão 3, use ViewBag.

3.3 View tipados

Para associar manualmente uma classe a um view, basta definir o atributo Inherits da diretiva @ Page:

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
    Inherits="System.Web.Mvc.ViewPage<AppCapitulo3.Models.Paises>" %>
```

Os membros da classe Paises associada ao view são acessados por intermédio do objeto Model:

```
<%:Model.Nome %>
```

Observe a figura 3.3:

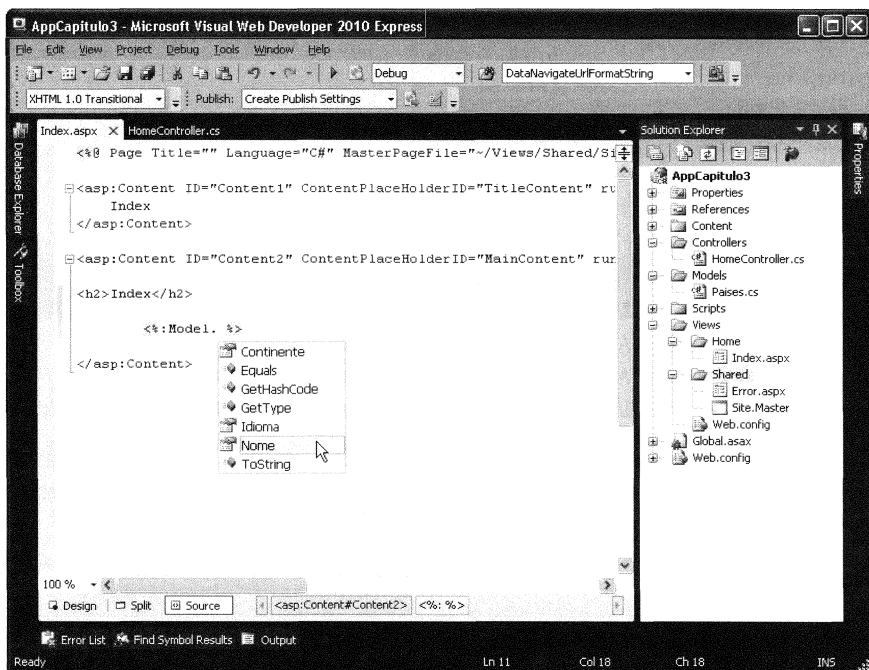


Figura 3.3 – Propriedades da classe Paises acessadas com o auxílio do objeto Model.

Trechos de códigos são executados com o delimitador <% %>:

```
<%foreach (var item in Model) {%>
    // Restante do código aqui
<%}%>
```

Exemplo:

```
<ul>
  <%foreach (var item in Model) {%>
    <li>
      <%=item.Nome%>
    </li>
  <%}%>
</ul>
```

3.4 View predefinidos

O ASP.NET MVC possui diversos view predefinidos que podem ser usados para exibir listas de itens, detalhes de um registro específico, inserir, atualizar e excluir registros.

3.4.1 Template List

Para exibir uma lista de itens, definimos inicialmente um método de controlador na classe controladora HomeController:

```
using System.Collections.Generic;
using System.Linq;
using System.Web.Mvc;
using AppCapitulo3.Models;

namespace AppCapitulo3.Controllers {
    public class HomeController : Controller {
        List<Países> lst = new List<Países> {
            new Países { Nome="Brasil", Continente="América do sul", Idioma="Português" },
            new Países { Nome="Alemanha", Continente="Europa", Idioma="Alemão" },
            new Países { Nome="Austrália", Continente="Oceania", Idioma="Inglês" },
            new Países { Nome="Uruguai", Continente="América do sul", Idioma="Espanhol" },
            new Países { Nome="Japão", Continente="Ásia", Idioma="Japonês" }
        };

        public ActionResult Index() {
            return View();
        }

        public ActionResult List() {
            return View(lst);
        }
    }
}
```

Em seguida, criamos o view predefinido definindo a opção **Scaffold template** na caixa de diálogo **Add View** como **List**. Observe a figura 3.4:

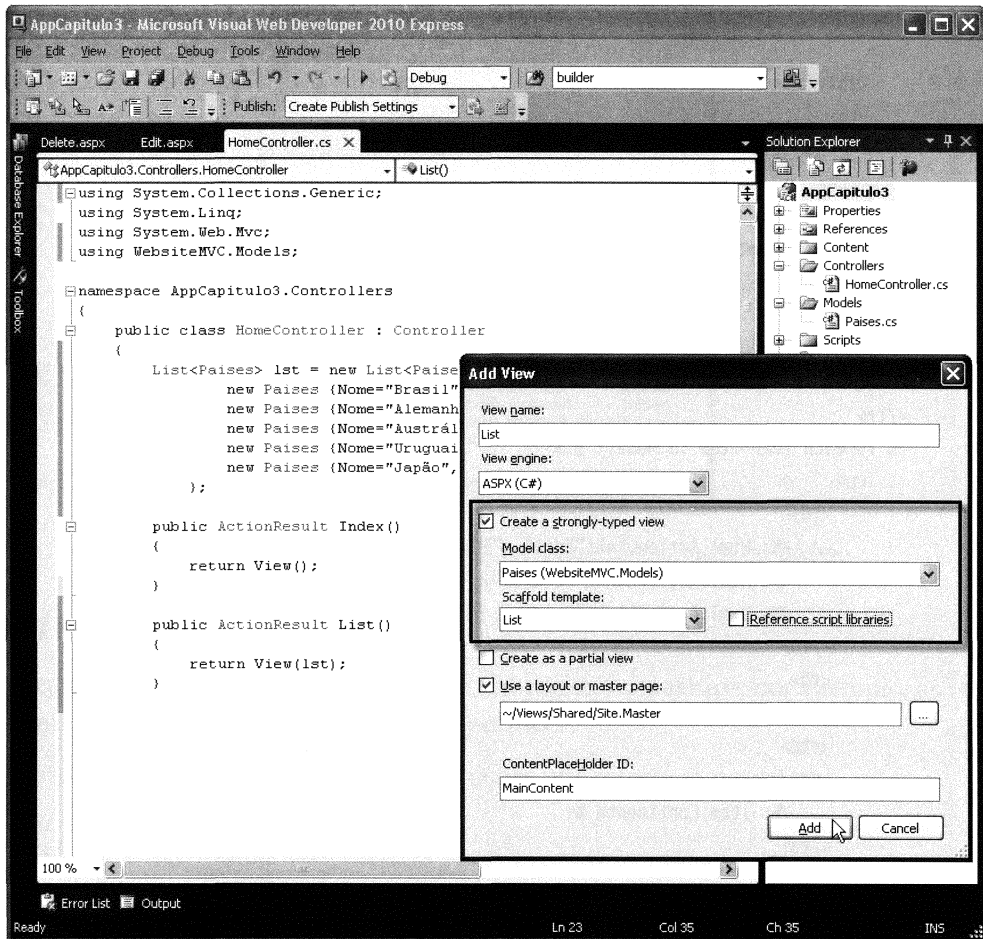


Figura 3.4 – Criando o View e o template List.

Quando clicamos em **Add**, é gerado o código a seguir:

```
<% Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Website.Master"
    Inherits="System.Web.Mvc.ViewPage<IEnumerable<AppCapitulo3.Models.Paises>>" %>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
    List
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
    <h2>List</h2>
    <p>
        <%: Html.ActionLink("Create New", "Create") %>
    </p>
```

```

<table>
  <tr>
    <th></th>
    <th>
      Nome
    </th>
    <th>
      Continente
    </th>
    <th>
      Idioma
    </th>
  </tr>
  <% foreach (var item in Model) { %>
    <tr>
      <td>
        <%= Html.ActionLink("Edit", "Edit", new { pais = item.Nome })%> |
        <%= Html.ActionLink("Details", "Details", new { pais=item.Nome })%> |
        <%= Html.ActionLink("Delete", "Delete", new { pais = item.Nome })%>
      </td>
      <td>
        <%= item.Nome %>
      </td>
      <td>
        <%= item.Continente %>
      </td>
      <td>
        <%= item.Idioma %>
      </td>
    </tr>
  <% } %>
</table>
</asp:Content>

```

Para tornar o exemplo funcional, substituímos as linhas:

```

<%= Html.ActionLink("Edit", "Edit", new { /* id=item.PrimaryKey */ }) %> |
<%= Html.ActionLink("Details", "Details", new { /* pais=item.PrimaryKey */ })%> |
<%= Html.ActionLink("Delete", "Delete", new { /* id=item.PrimaryKey */ }) %>

```

Por:

```

<%= Html.ActionLink("Edit", "Edit", new { pais = item.Nome })%> |
<%= Html.ActionLink("Details", "Details", new { pais=item.Nome })%> |
<%= Html.ActionLink("Delete", "Delete", new { pais = item.Nome })%>

```

Ao executar o exemplo, temos a saída apresentada pela figura 3.5:

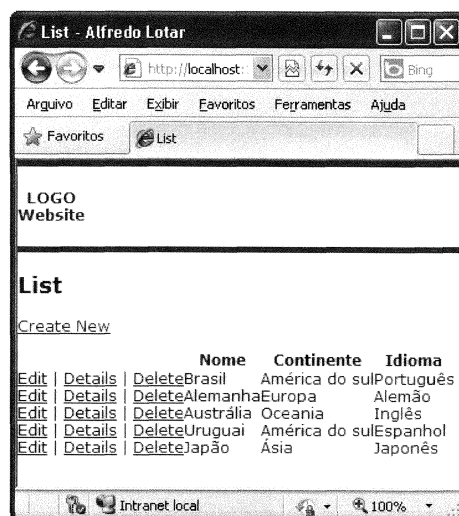


Figura 3.5 – Template List exibido no navegador com formatação-padrão.

3.4.2 Template Details

O template Details exibe informações de um registro específico, por exemplo, as informações de cadastro de um usuário específico.

O primeiro passo é a definição das instruções que retornam o registro:

```
public ActionResult Details(string pais) {
    if (pais == null) return View("NotFound");
    var model = (from p in lst.Where(x => x.Nome == pais)
        select p).FirstOrDefault();
    if (model == null) return View("NotFound");
    return View(model);
}
```

O exemplo usa o view NotFound para exibir uma mensagem de erro. Crie-o no diretório Views\Shared:

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
    Inherits="System.Web.Mvc.ViewPage<dynamic>" %>
<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
    NotFound
</asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
    <h2>NotFound</h2>
    <p>Registro não encontrado.</p>
</asp:Content>
```

Em seguida, geramos o view com o template Details. Observe a figura 3.6:

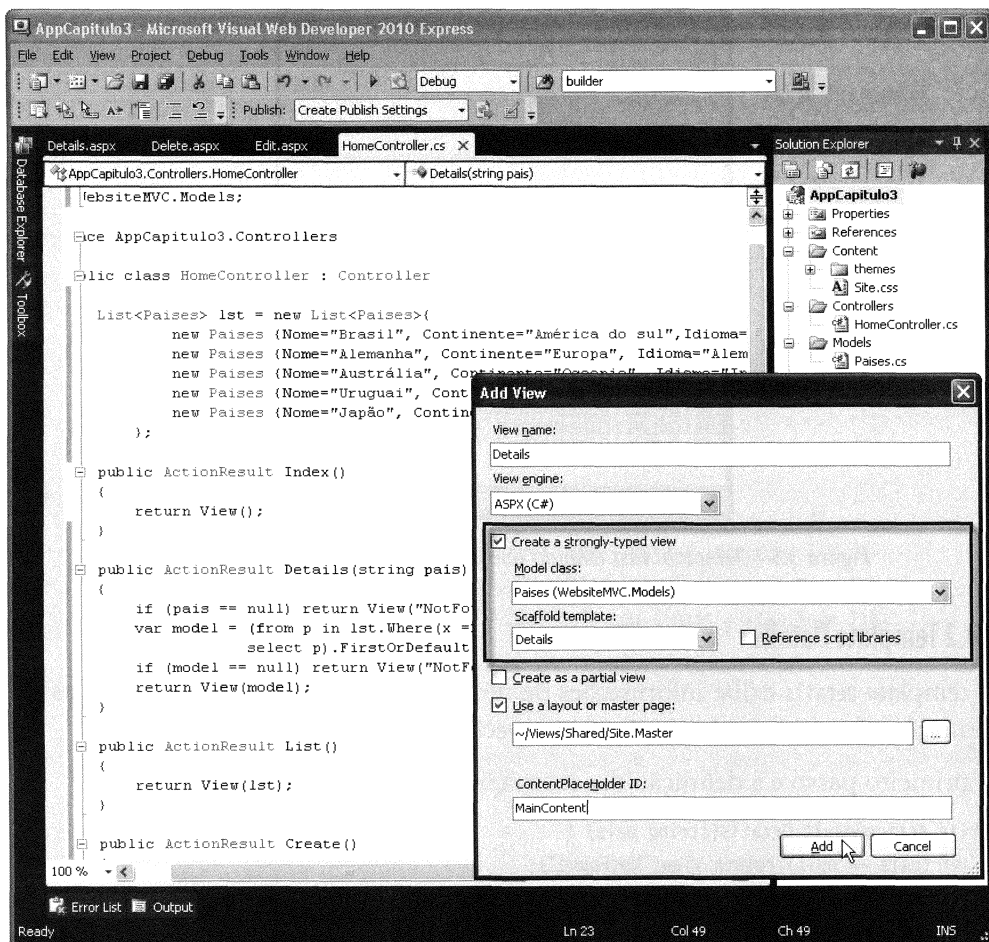


Figura 3.6 – Criando o view e o template Details.

Quando clicamos no botão **Add** da caixa de diálogo **Add View**, geramos o código a seguir:

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
    Inherits="System.Web.Mvc.ViewPage<AppCapitulo3.Models.Paises>" %>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
    Details
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
    <h2>Details</h2>

    <fieldset>
        <legend>Países</legend>
        <div class="display-label">Nome</div>
        <div class="display-field"><%= Model.Nome %></div>
```

```

<div class="display-label">Continente</div>
<div class="display-field"><%= Model.Continente %></div>

<div class="display-label">Idioma</div>
<div class="display-field"><%= Model.Idioma %></div>
</fieldset>
<p>
  <%= Html.ActionLink("Edit", "Edit", new { /* id=Model.PrimaryKey */ }) %> |
  <%= Html.ActionLink("Back to List", "Index") %>
</p>
</asp:Content>

```

Ao executar o exemplo, temos a saída apresentada na figura 3.7:

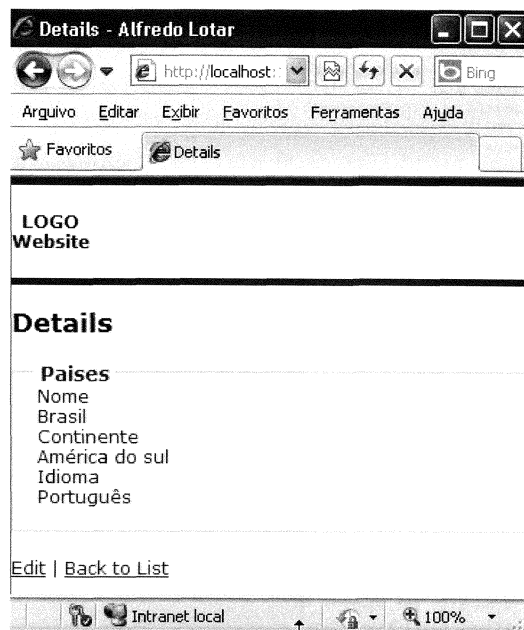


Figura 3.7 – Detalhes de um registro com o template Details.

3.4.3 Template Create

Para inserir um registro em um banco de dados podemos gerar um template, como o template Create.

Novamente, começamos com o código criando os métodos de controlador apropriados:

```

public ActionResult Create() {
    return View();
}
[HttpPost]

```

```

public ActionResult Create(Paises p) {
    if (!ModelState.IsValid) return View();
    // Aqui o código que grava o registro
    return RedirectToAction("Index");
}

```

Em seguida, criamos o view e definimos o tipo de template e a classe associada. Observe a figura 3.8:

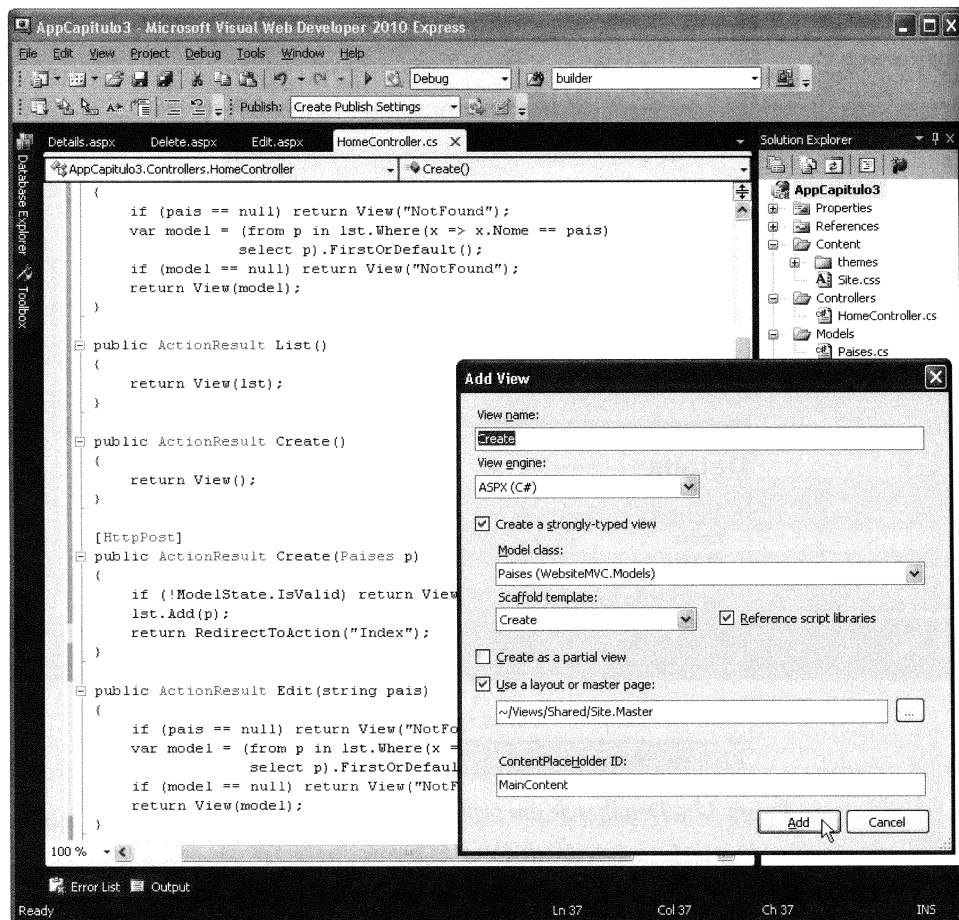


Figura 3.8 – Criando o view e o template Create.

O código gerado para view é o seguinte:

```

<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
    Inherits="System.Web.Mvc.ViewPage<AppCapitulo3.Models.Paises>" %>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
    Create
</asp:Content>

```

```

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
    <h2>Create</h2>
    <script src="<%= Url.Content("~/Scripts/jquery.validate.min.js") %>"
        type="text/javascript"></script>
    <script src="<%= Url.Content("~/Scripts/jquery.validate.unobtrusive.min.js") %>"
        type="text/javascript"></script>
    <% using (Html.BeginForm()) { %>
        <%= Html.ValidationSummary(true) %>
        <fieldset>
            <legend>Países</legend>

            <div class="editor-label">
                <%= Html.LabelFor(model => model.Nome) %>
            </div>
            <div class="editor-field">
                <%= Html.EditorFor(model => model.Nome) %>
                <%= Html.ValidationMessageFor(model => model.Nome) %>
            </div>

            <div class="editor-label">
                <%= Html.LabelFor(model => model.Continente) %>
            </div>
            <div class="editor-field">
                <%= Html.EditorFor(model => model.Continente) %>
                <%= Html.ValidationMessageFor(model => model.Continente) %>
            </div>

            <div class="editor-label">
                <%= Html.LabelFor(model => model.Idioma) %>
            </div>
            <div class="editor-field">
                <%= Html.EditorFor(model => model.Idioma) %>
                <%= Html.ValidationMessageFor(model => model.Idioma) %>
            </div>
            <p>
                <input type="submit" value="Create" />
            </p>
        </fieldset>
    <% } %>
    <div>
        <%= Html.ActionLink("Back to List", "Index") %>
    </div>
</asp:Content>

```

O código anterior gera a saída exibida na figura 3.9:



Figura 3.9 – Template Create exibido no navegador.

3.4.4 Template Edit

O template Edit é gerado da mesma forma que os demais apresentados neste livro. Primeiramente, criamos o código:

```
public ActionResult Edit(string pais) {
    if (pais == null) return View("NotFound");
    var model = (from p in lst.Where(x => x.Nome == pais) select p).FirstOrDefault();
    if (model == null) return View("NotFound");
    return View(model);
}
```

```
[HttpPost]
public ActionResult Edit(Paises p) {
    if (!ModelState.IsValid) return View();
    // Aqui o código que grava o registro
    return RedirectToAction("Index");
}
```

Em seguida, o view a partir do método Edit. Observe a figura 3.10:

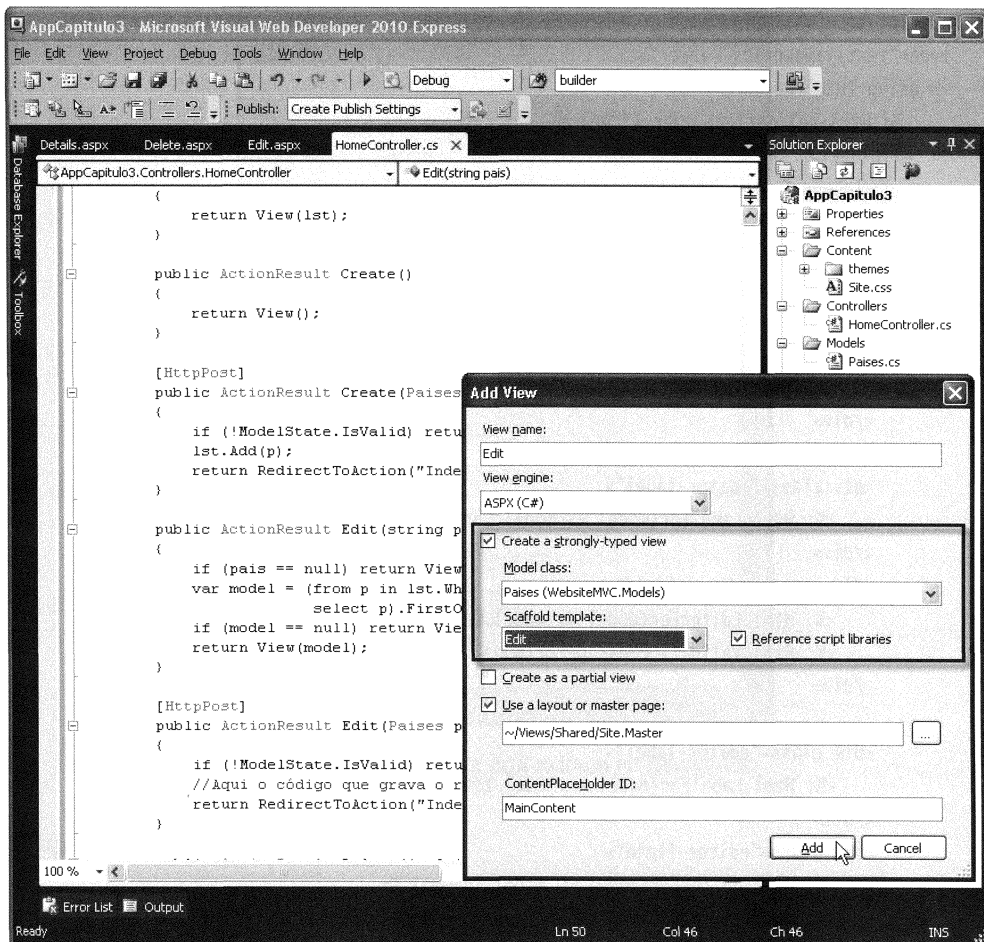


Figura 3.10 – Criando o view e o template Edit.

Quando clicamos em Add, geramos o código a seguir:

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
    Inherits="System.Web.Mvc.ViewPage<AppCapitulo3.Models.Paises>" %>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
    Edit
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
    <h2>Edit</h2>
    <script src="<%= Url.Content("~/Scripts/jquery.validate.min.js") %>"
        type="text/javascript"></script>
```

```

<script src="%: Url.Content("~/Scripts/jquery.validate.unobtrusive.min.js") %>"
type="text/javascript"></script>
<% using (Html.BeginForm()) { %>
    <%: Html.ValidationSummary(true) %>
    <fieldset>
        <legend>Países</legend>
        <div class="editor-label">
            <%: Html.LabelFor(model => model.Nome) %>
        </div>
        <div class="editor-field">
            <%: Html.EditorFor(model => model.Nome) %>
            <%: Html.ValidationMessageFor(model => model.Nome) %>
        </div>

        <div class="editor-label">
            <%: Html.LabelFor(model => model.Continente) %>
        </div>
        <div class="editor-field">
            <%: Html.EditorFor(model => model.Continente) %>
            <%: Html.ValidationMessageFor(model => model.Continente) %>
        </div>

        <div class="editor-label">
            <%: Html.LabelFor(model => model.Idioma) %>
        </div>
        <div class="editor-field">
            <%: Html.EditorFor(model => model.Idioma) %>
            <%: Html.ValidationMessageFor(model => model.Idioma) %>
        </div>
    <p>
        <input type="submit" value="Save" />
    </p>
</fieldset>
<% } %>
<div>
    <%: Html.ActionLink("Back to List", "Index") %>
</div>
</asp:Content>

```

A saída gerada vemos na figura 3.11:

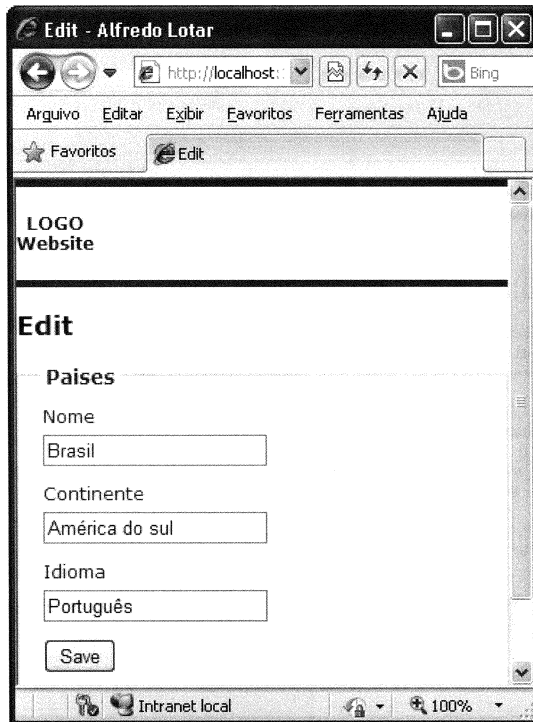


Figura 3.11 – Template Edit exibido no navegador.

3.4.5 Template Delete

Para excluir um registro por intermédio do template Delete, primeiramente, defina o código necessário:

```
public ActionResult Delete(string pais) {
    if (pais == null) return View("NotFound");
    var model = (from p in lst.Where(x => x.Nome == pais) select p).FirstOrDefault();
    if (model == null) return View("NotFound");
    return View(model);
}

[HttpPost]
public ActionResult Delete(Paises p) {
    if (!ModelState.IsValid) return View();
    // Aqui o código que exclui o registro
    return RedirectToAction("Index");
}
```

Em seguida, o template é gerado conforme a figura 3.12:

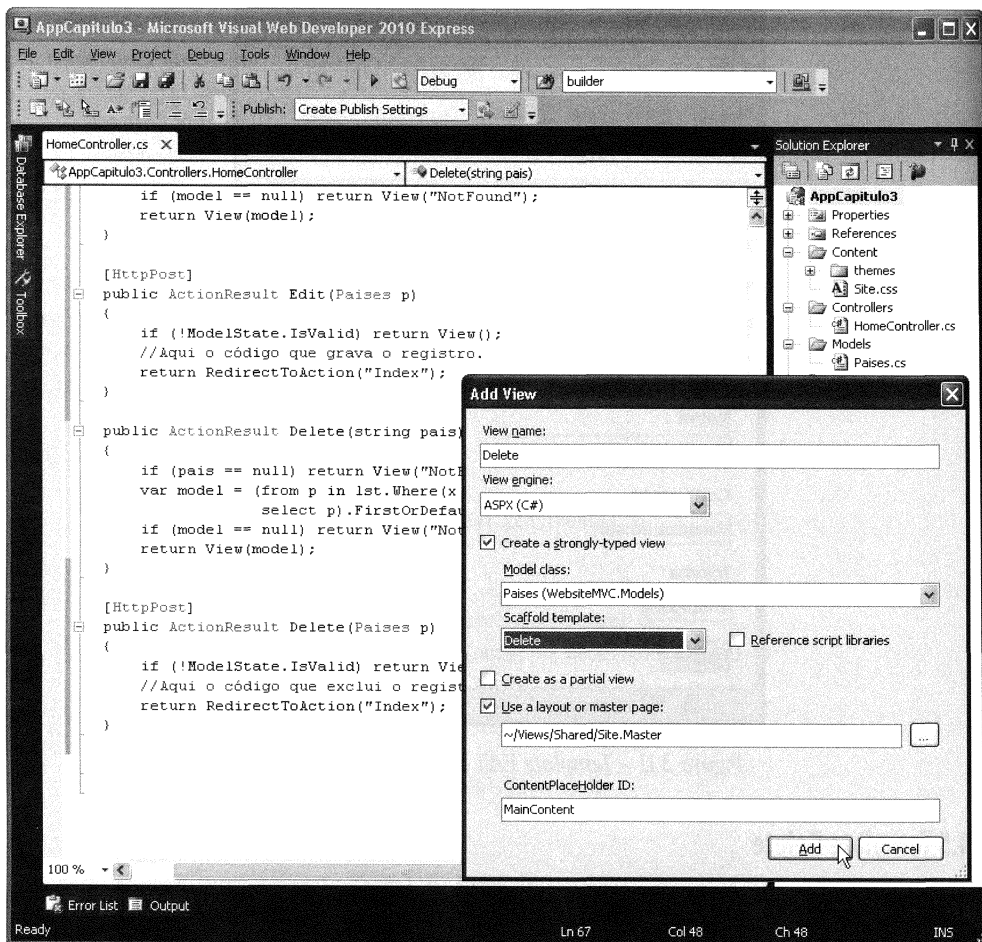


Figura 3.12 – Criando o view e o template Delete.

Ao clicar em **Add**, geramos o seguinte código:

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
    Inherits="System.Web.Mvc.ViewPage<AppCapitulo3.Models.Paises>" %>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
    Delete
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
    <h2>Delete</h2>
    <h3>Are you sure you want to delete this?</h3>
    <fieldset>
        <legend>Países</legend>
```

```

<div class="display-label">Nome</div>
<div class="display-field"><%: Model.Nome %></div>

<div class="display-label">Continente</div>
<div class="display-field"><%: Model.Continente %></div>

<div class="display-label">Idioma</div>
<div class="display-field"><%: Model.Idioma %></div>
</fieldset>
<% using (Html.BeginForm()) { %>
    <p>
        <input type="submit" value="Delete" /> |
        <%: Html.ActionLink("Back to List", "Index") %>
    </p>
<% } %>
</asp:Content>

```

A seguir, na figura 3.13, temos a saída gerada:

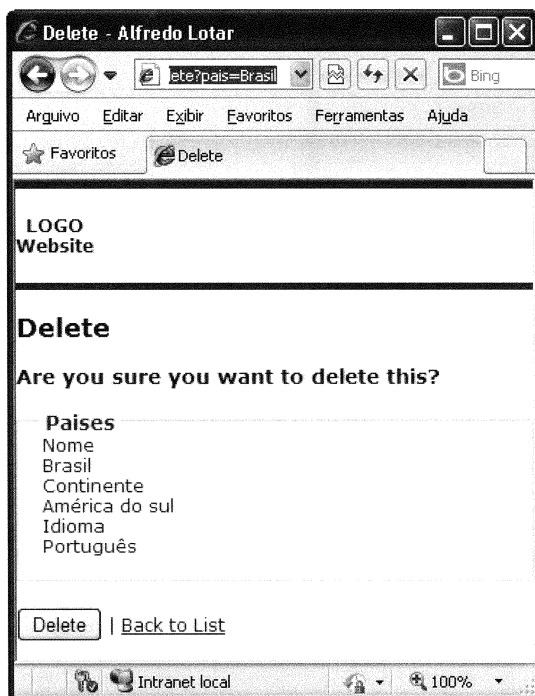


Figura 3.13 – Template Delete exibido no navegador.

3.4.6 Criando ou alterando templates

Para alterar a aparência-padrão dos templates (modelos predefinidos), acesse:

C:\Arquivos de programas\Microsoft Visual Studio 10.0\Common7\IDE\WDEExpress\ItemTemplates\CSharp\Web\MVC 3\CodeTemplates\AddView\AspxCSharp

Obs.: o caminho para os arquivos .tt varia muito, pois depende da versão do Windows, Visual Studio instalada.

Altere os templates-padrão – extensão .tt ou crie os novos. Observe a figura 3.14:

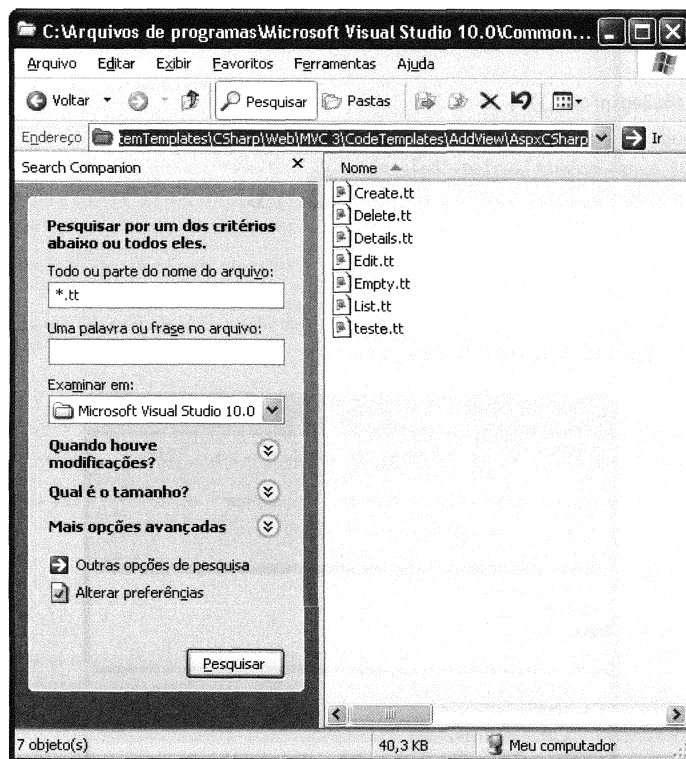


Figura 3.14 – Arquivos dos modelos predefinidos.

3.5 Partial view

O ASP.NET MVC usa as chamadas partial view, ou seja, um user control. O mecanismo de exibição ASPX usa user controls com a extensão .ascx. Mas outros mecanismos de exibição, como o Razor, podem empregar outras extensões.

3.5.1 Acrescentando um novo partial view

Há dois modos de se acrescentar um novo view a um projeto ASP.NET MVC. No primeiro modo, clicamos com o botão direito do mouse em um subdiretório qualquer do diretório Views, em seguida, apontamos para **Add** e clicamos em **New Item....**

Observe a figura 3.15:

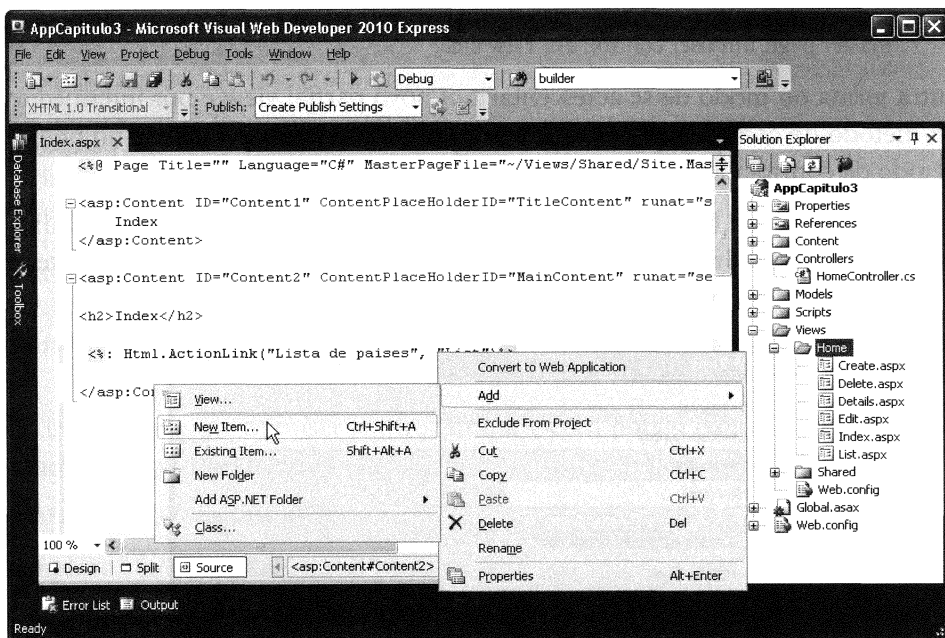


Figura 3.15 – Acrescentando um novo partial view.

Na caixa de diálogo **Add New Item**, selecionamos **MVC 4 View User Control (ASPX)** e determinamos o nome do user control na caixa de texto **Name**. Observe a figura 3.16:

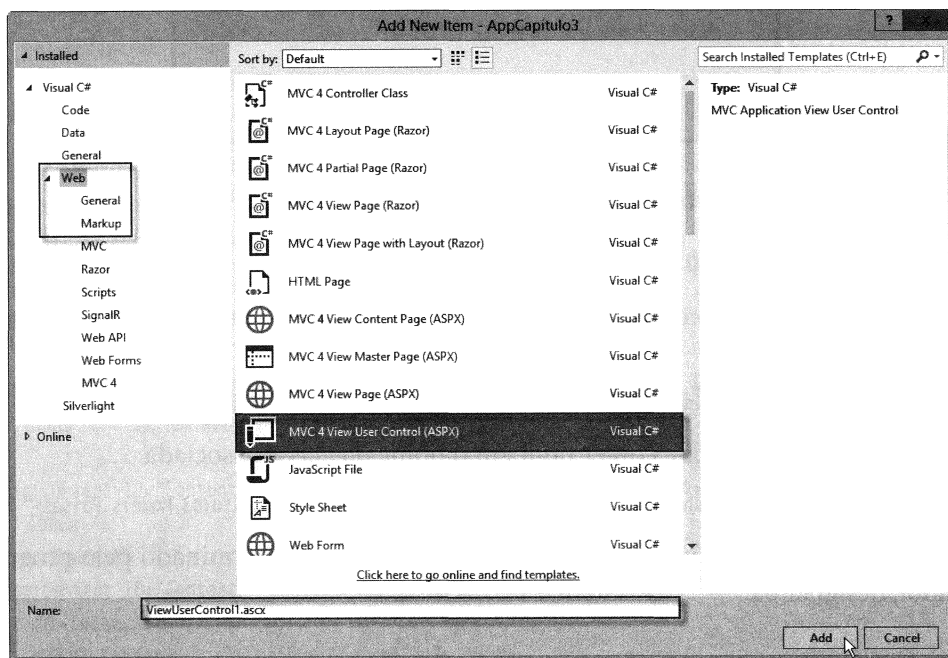


Figura 3.16 – Caixa de diálogo Add New Item.

Obs.: repare que, a partir da caixa de diálogo **Add New Item**, você pode adicionar uma master page, um user control, um view etc.

Outra forma ou modo de se acrescentar um novo partial view é por intermédio de um método de controlador. Você clica com o botão direito do mouse no nome do método de controlador; em seguida, seleciona e clica em **Add View...**

Na caixa de diálogo **Add View** você define o nome do user control e marca a caixa de seleção **Create as a partial view**. Se necessário, associe uma classe e defina um template para o user control. Observe a figura 3.17:

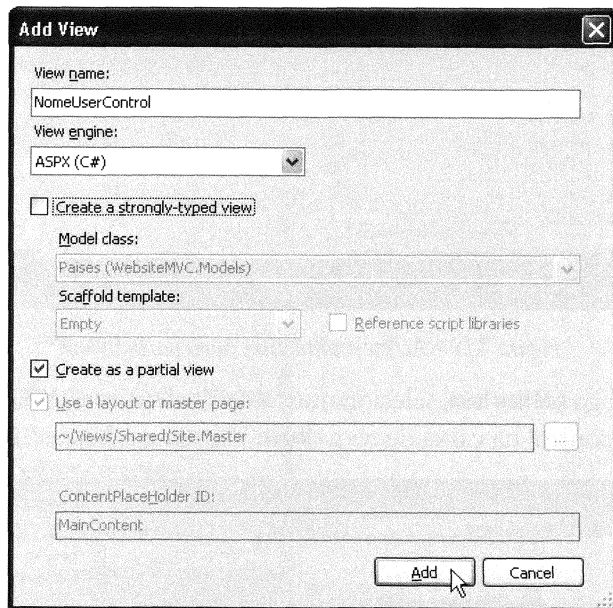


Figura 3.17 – Caixa de diálogo Add View.

3.5.2 Exibindo o conteúdo de um partial view

Quando um novo user control (partial view) é adicionado ao projeto, ele possui uma única linha de código:

```
<%@ Control Language="C#" Inherits="System.Web.Mvc.ViewUserController<dynamic>" %>
```

Claro que essa linha pode conter também o nome da classe associada:

```
<%@ Control Language="C#" Inherits="System.Web.Mvc.ViewUserController<AppCapitulo3.Models.Paises>" %>
```

As linhas seguintes definem o conteúdo do user control determinado pelo programador, designer etc.

Exemplo:

```
<%@ Control Language="C#" Inherits="System.Web.Mvc.ViewUserControl<dynamic>" %>
<ul>
    <li>Home</li>
    <li>Eletrônica</li>
    <li>Idiomas</li>
    <li>Negócios</li>
    <li>Programação</li>
    <li>Segurança</li>
    <li>UML</li>
</ul>
```

Para invocar e exibir um user control simples, com pouco conteúdo, em um view, usamos o método `Partial`:

```
<%:Html.Partial("usrUserControl")%>
```

Exemplo:

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
    Inherits="System.Web.Mvc.ViewPage<dynamic>" %>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
    Index
</asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
    <h2>Index</h2>
    <%:Html.Partial("usrUserControl")%>
</asp:Content>
```

O método `Partial` armazena o conteúdo na memória e o exibe como uma string. O método `RenderPartial` é mais eficiente, pois envia o conteúdo diretamente para a saída. Ideal para user controls com muito conteúdo ou imagens:

```
<%Html.RenderPartial("usrUserControl");%>
```

Exemplo:

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
    Inherits="System.Web.Mvc.ViewPage<dynamic>" %>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
    Index
</asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
    <h2>Index</h2>
    <%Html.RenderPartial("usrUserControl");%>
</asp:Content>
```

Em um view Razor, use:

```
@{Html.RenderPartial("usrUserControl");}
```

Se preferir, use um método de controlador para exibir o conteúdo de um user control. Por exemplo, o método Action acessa o método de controlador Menu:

```
<%:Html.Action("Menu")%>
```

Que, por sua vez, carrega o user control `usrUserControl.ascx`:

```
[ChildActionOnly]
public ActionResult Menu() {
    return PartialView("usrUserControl");
}
```

Se o método de controlador aceita ou requer parâmetros, defina-os:

```
<%:Html.Action("Menu", new { id = 5 })%>
```

O conteúdo de um user control pode ser enviado diretamente para a saída por intermédio do método `RenderAction`:

```
<%Html.RenderAction("Menu");%>
```

Repare que o método `RenderAction` também acessa o método de controlador `Menu`:

```
[ChildActionOnly]
public ActionResult Menu() {
    return PartialView("usrUserControl");
}
```

Em um view Razor, use:

```
@{Html.RenderAction("Menu");}
```

3.5.3 Associando classes a um user control

Aplicações que acessam banco de dados podem dividir o conteúdo de um view em um ou mais user controls. Por exemplo, parte do conteúdo do view a seguir:

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
    Inherits="System.Web.Mvc.ViewPage<IEnumerable<AppCapitulo3.Models.Paises>>" %>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
    Lista
</asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
    <h2>Lista</h2>
```



```

<table>
  <% foreach (var item in Model) { %>
    <tr>
      <td>
        <%= item.Nome%>
      </td>
      <td>
        <%= item.Continente%>
      </td>
      <td>
        <%= item.Idioma%>
      </td>
    </tr>
  <% } %>
</table>
</asp:Content>

```

Pode ser extraído:

```

<tr>
  <td>
    <%= item.Nome%>
  </td>
  <td>
    <%= item.Continente%>
  </td>
  <td>
    <%= item.Idioma%>
  </td>
</tr>

```

E inserido, alterado e adaptado em um user control:

```

<%@ Control Language="C#" Inherits="System.Web.Mvc.ViewUserControl<AppCapitulo3.Models.Paises>" %>
<tr>
  <td>
    <%= Model.Nome%>
  </td>
  <td>
    <%= Model.Continente%>
  </td>
  <td>
    <%= Model.Idioma%>
  </td>
</tr>

```

Para invocar o user control, usamos o método `RenderPartial`:

```
<%Html.RenderPartial("ListUserController", item); %>
```

Exemplo:

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
    Inherits="System.Web.Mvc.ViewPage<IEnumerable<AppCapitulo3.Models.Paises>>" %>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
    Lista
</asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
    <h2>Lista</h2>
    <table>
        <% foreach (var item in Model) {%>
            <%Html.RenderPartial("ListUserController", item); %>
        <% } %>
    </table>
</asp:Content>
```

Model

O model ou modelo contém as classes e arquivos da camada de dados. É no model que você define a lógica de acesso de dados, as interfaces e as classes que as implementam.

O model contém também as classes do ADO.NET Entity Framework e as classes que definem as regras de validação e de negócios da aplicação.

4.1 Desenvolvendo uma nova aplicação

Inicie o Visual Studio. Acesse o menu **File** e clique em **New Project ...**.

Na caixa de diálogo **New Project**, selecione **Web > Visual Studio 2012 > ASP.NET MVC 4 Web Application**. Nomeie o projeto como **AppCapitulo4**. Clique em **OK**.

Copie o arquivo **Content/Site.css** e a master page **Views/Shared/Site.Master** da aplicação **AppCapitulo1** para a aplicação **AppCapitulo4**.

Os exemplos deste livro usam o banco de dados **Northwind**. Para criá-lo, execute o arquivo **CriarNorthwindBD.bat** localizado nos arquivos de exemplo deste livro, ou execute os arquivos **Northwind.sql** e **Createdatabase.sql** no **SQL Server® 2008 Management Studio**.

4.2 Usando ADO.NET

Nosso exemplo com ADO.NET consiste em uma lista de categorias. Exibiremos os campos **CategoryID**, **CategoryName**, **Description** da tabela **Categories** do banco de dados **Northwind**.

Primeiramente, criamos um arquivo **.cs** no diretório **Models** da aplicação **ASP.NET MVC**; em seguida, adicionamos as classes da camada de dados.

Observe o código a seguir:

```
using System;  
using System.Collections.Generic;  
using System.Configuration;  
using System.Data;
```

```

using System.Data.SqlClient;
using System.Web.Configuration;

namespace AppCapitulo4.Models {
    public class CategoriesRepository {
        public List<Categorie> GetAllCategories() {
            SqlConnectionSettings getString = WebConfigurationManager.ConnectionStrings["nwind"]
                as SqlConnectionSettings;
            if (getString != null) {
                string sSql = "select CategoryID, CategoryName, Description from Categories";
                using (SqlConnection conn = new SqlConnection(getString.ConnectionString)) {
                    List<Categorie> lst = new List<Categorie>();

                    SqlDataReader r = null;
                    SqlCommand cmd = new SqlCommand(sSql, conn);
                    conn.Open();
                    r = cmd.ExecuteReader(CommandBehavior.CloseConnection);
                    if (r.HasRows) {
                        int categoryID = r.GetOrdinal("CategoryID");
                        int categoryName = r.GetOrdinal("CategoryName");
                        int description = r.GetOrdinal("Description");

                        while (r.Read()) {
                            Categorie p = new Categorie();
                            p.CategoryID = r.GetInt32(categoryID);
                            p.CategoryName = r.GetString(categoryName).ToString();
                            p.Description = r.GetString(description).ToString();
                            lst.Add(p);
                        }
                    }
                    return lst;
                }
            }
            return null;
        }
    }

    public class Categorie {
        public int CategoryID { get; set; }
        public string CategoryName { get; set; }
        public string Description { get; set; }
    }
}

```

O código anterior não tem nenhum mistério. Preenchemos as propriedades da classe `Categorie` com um objeto `SqlDataReader`:

```
r = cmd.ExecuteReader(CommandBehavior.CloseConnection);
while (r.Read()) {
    Categorie p=new Categorie();
    p.CategoryID = r.GetInt32(categoryID);
    p.CategoryName = r.GetString(categoryName).ToString();
    p.Description = r.GetString(description).ToString();
    lst.Add(p);
}
```

E retornamos uma lista de objetos `Categorie`:

```
public List<Categorie> GetAllCategories() {...}
```

Em seguida, invocamos o método `GetAllCategories` da classe `CategoriesRepository` no controlador:

```
using System.Web.Mvc;
using AppCapitulo4.Models;
namespace AppCapitulo4.Controllers {
    public class CategorieController : Controller {
        CategoriesRepository _db = new CategoriesRepository();
        public ActionResult Index() {
            var model = _db.GetAllCategories();
            if (model == null)
                return View("NotFound");
            else
                return View(model);
        }
    }
}
```

Para criar o view `NotFound`, siga os passos descritos no tópico 3.4.2.

Compile a aplicação, conforme a figura 4.1:

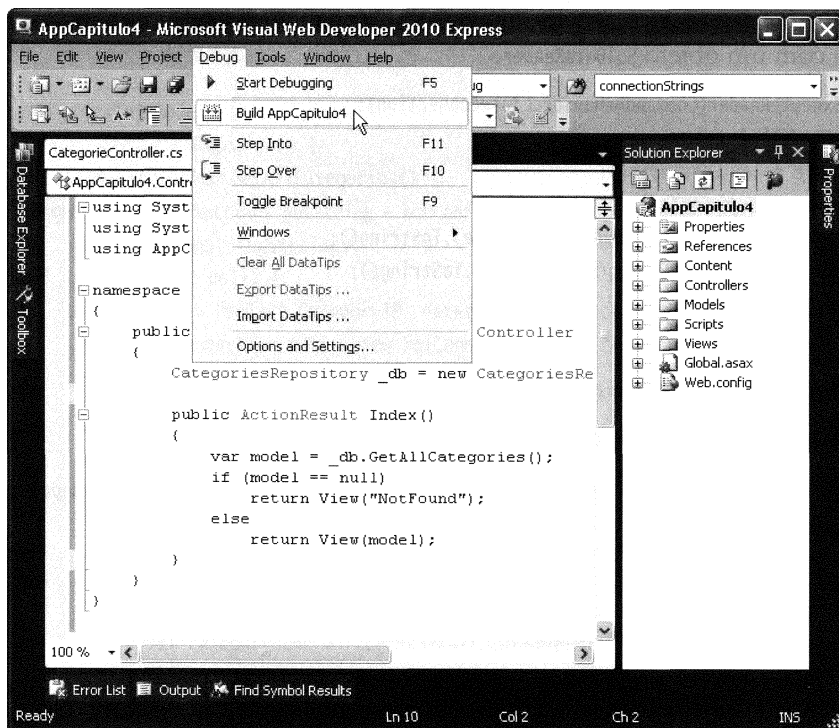


Figura 4.1 – Compilando a aplicação.

Para testar o exemplo, adicionamos um view. Observe a figura 4.2:

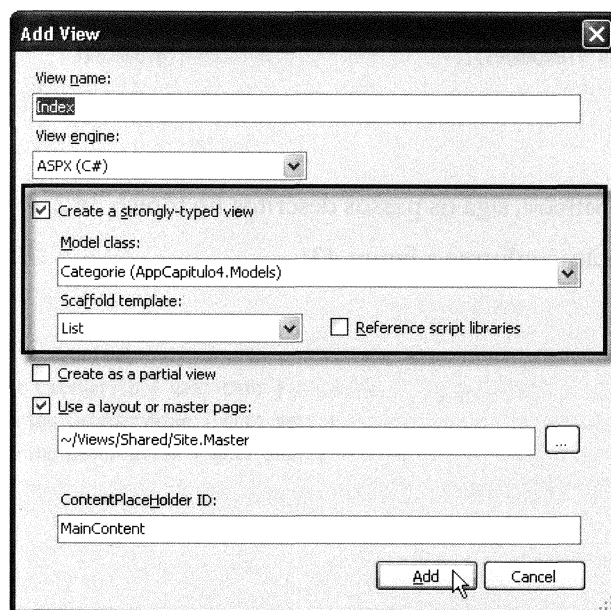


Figura 4.2 – Criando um novo view.

No arquivo web.config acrescentamos as linhas a seguir:

```
<connectionStrings>
  <add name="nwind" connectionString="Data Source=.\SQLEXPRESS;Integrated
    Security=SSPI;Initial Catalog=northwind;" />
</connectionStrings>
```

Após algumas alterações no view, pressione **F5** no Visual Studio, ou digite no navegador:

<http://localhost/AppCapitulo4/Categorie/index>

O resultado vemos na figura 4.3:

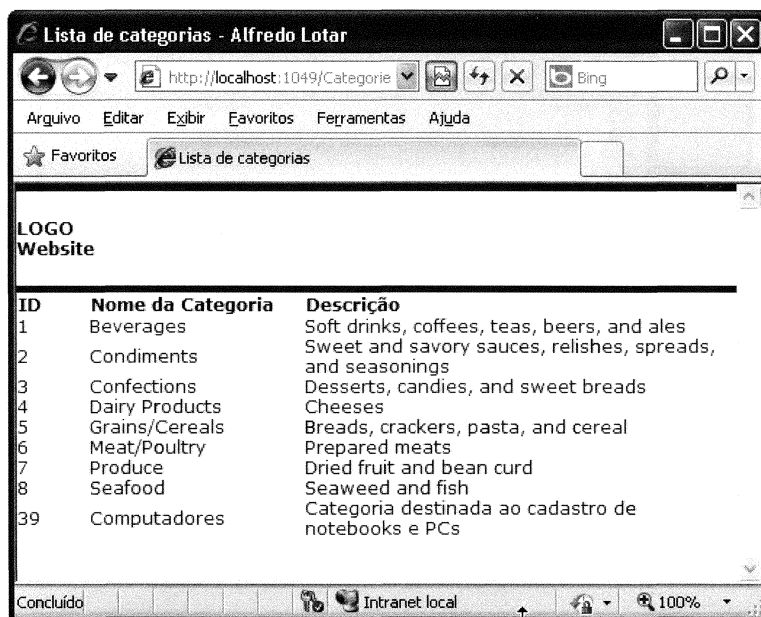


Figura 4.3 – Categorias armazenadas na tabela Categories.

4.3 ADO.NET Entity Framework

O ADO.NET Entity Framework permite que você programe usando entidades relacionadas criadas manualmente ou a partir de tabelas de um banco de dados.

Com o ADO.NET Entity Framework, é possível facilmente adicionar, atualizar, excluir e extrair informações de um banco de dados. Tudo isso com poucas linhas de código, comparado ao ADO.NET.

As entidades do ADO.NET Entity Framework podem ser visualizadas graficamente a partir de um arquivo .edmx.

Para adicionar entidades para um arquivo .edmx a partir do banco de dados Northwind, siga os passos descritos a seguir:

Clique com o botão direito do mouse no diretório *Models* na janela **Solution Explorer**; em seguida, aponte para **Add** e clique em **New Item**.

Selecione o template ADO.NET Entity Data Model.

Digite o nome do arquivo (<nome do arquivo>.edmx) e clique em **Add**. Defina a linguagem de programação como Visual C#. Observe a figura 4.4:

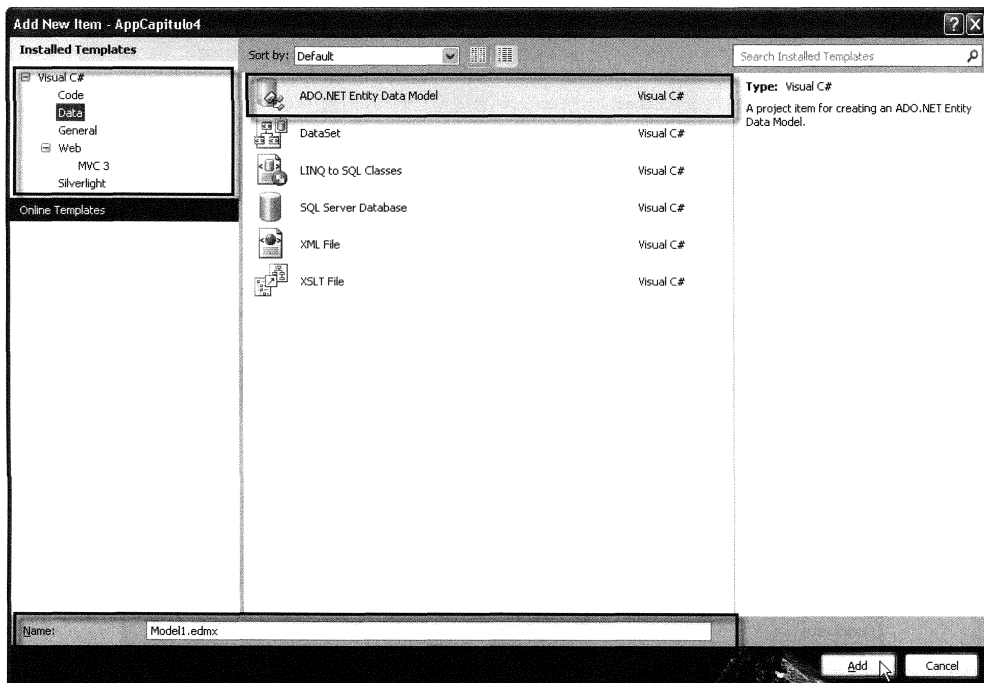


Figura 4.4 – Caixa de diálogo Add New Item.

Selecione **Generate from database** na caixa de diálogo **Choose Model Contents** e clique em **Next**. Observe a figura 4.5:

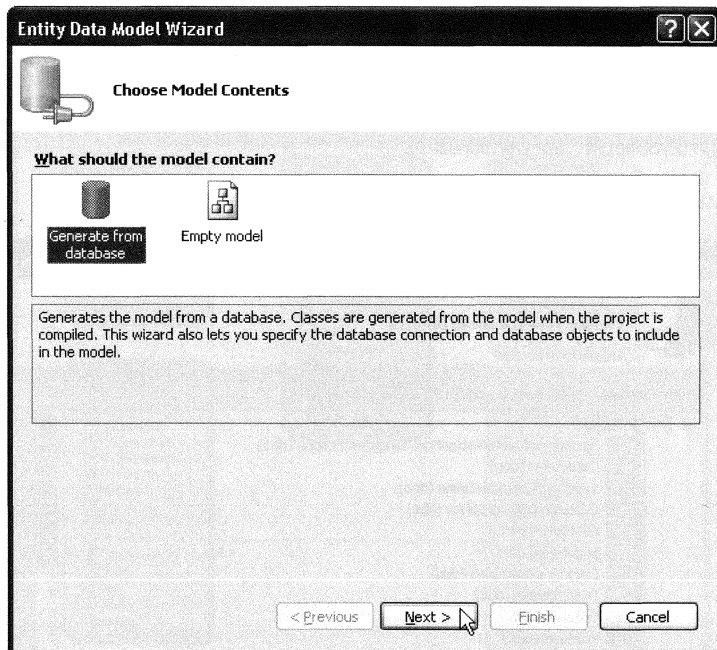


Figura 4.5 – Caixa de diálogo Choose Model Contents.

Selecione o banco de dados Northwind e defina o nome da string de conexão como NorthwindBD, conforme observamos na figura 4.6:

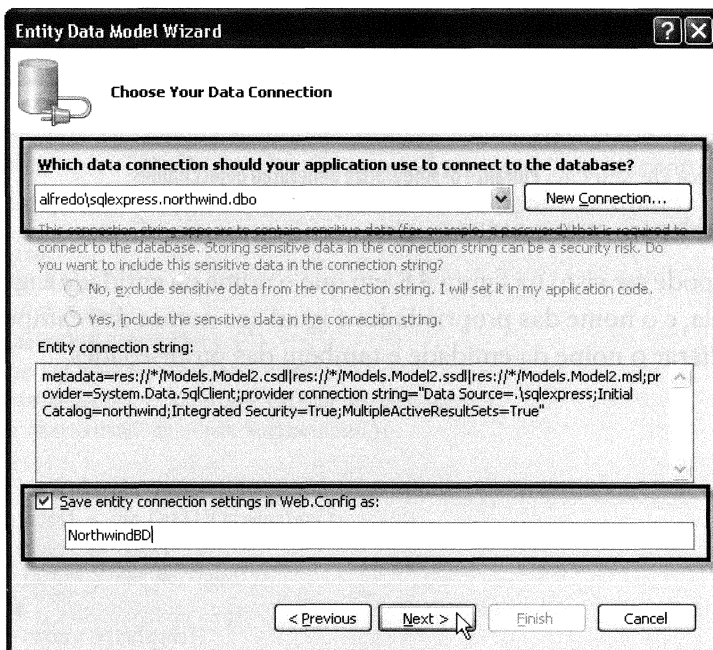


Figura 4.6 – Configurando a conexão com o banco de dados.

Clique em **Next**. Inicie o servidor SQL antes de clicar em **Next**.

Na caixa de diálogo **Choose Your Database Objects**, selecione as tabelas: Categories, Products, Customers, Orders, Order_Details. Defina também o nome de uma namespace (exemplo: AlfredoLotar.Livro.AspNetMvc.NorthwindModel).

Observe a figura 4.7:

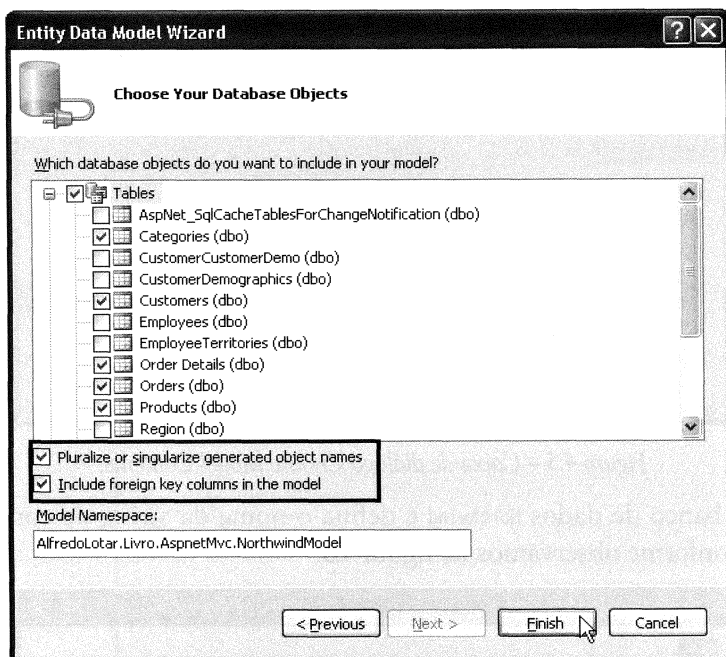


Figura 4.7 – Caixa de diálogo *Choose Your Database Objects*.

Marque a caixa de seleção **Pluralize or singularize generated object names**.

Clique em **Finish** para criar o arquivo .edmx.

O resultado pode ser visto na figura 4.8, em que o nome da entidade é igual ao nome de cada tabela, e o nome das propriedades é igual aos nomes dos campos da tabela. Você pode alterar o nome da entidade e também das propriedades.

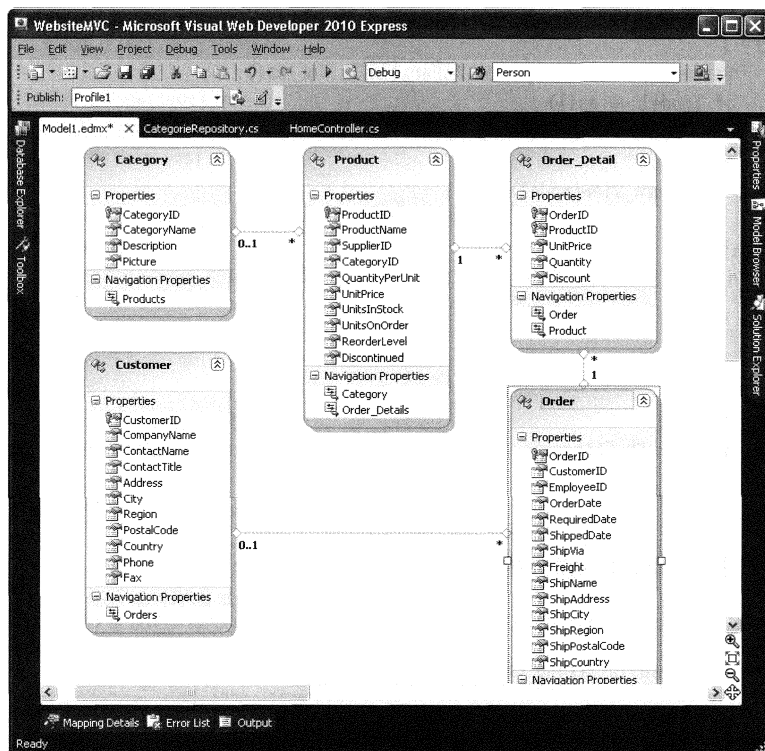


Figura 4.8 – Arquivo Model1.edmx aberto no Entity Designer.

4.3.1 Exibindo registros

Para exibir todas as categorias da tabela Categories, usamos o objeto Entity Framework NorthwindBD e instruções LINQ. O código é executado diretamente no controlador. Opção recomendada apenas em aplicações pequenas ou de teste. Exemplo:

```
using System.Linq;
using System.Web.Mvc;
using AppCapitulo4.Models;
namespace AppCapitulo4.Controllers {
    public class CategorieController : Controller {
        CategoriesRepository _db = new CategoriesRepository();
        private NorthwindBD db = new NorthwindBD();

        public ActionResult Index() {
            var model = _db.GetAllCategories();
            if (model == null)
                return View("NotFound");
            else
                return View(model);
        }
    }
}
```

```

public ActionResult List() {
    try {
        var model = db.Categories.AsParallel().ToList();
        if (model == null)
            return View("NotFound");
        else
            return View(model);
    }
    finally {
        if (db != null) db.Dispose();
    }
}
}

```

A linha `var model = db.Categories.AsParallel().ToList();` retorna uma lista genérica de categorias.

4.3.2 Exibindo detalhes de um registro

Para exibir um registro específico, filtramos as informações com a cláusula `where`:

```

public ActionResult Details(string categoria) {
    try {
        ViewBag.Message = categoria;
        var model = (from p in db.Products
                     where p.Category.CategoryName == categoria
                     select p).ToList();
        if (model == null)
            return View("NotFound");
        else
            return View(model);
    }
    finally {
        if (db != null) db.Dispose();
    }
}

```

Exemplo:

```

using System.Linq;
using System.Web.Mvc;
using AppCapitulo4.Models;

namespace AppCapitulo4.Controllers {
    public class CategorieController : Controller {
        CategoriesRepository _db = new CategoriesRepository();
        private NorthwindBD db = new NorthwindBD();
        public ActionResult Index() {
            var model = _db.GetAllCategories();

```

```

        if (model == null)
            return View("NotFound");
        else
            return View(model);
    }
    public ActionResult List() {
        try {
            var model = db.Categories.AsParallel().ToList();
            if (model == null)
                return View("NotFound");
            else
                return View(model);
        }
        finally {
            if (db != null) db.Dispose();
        }
    }
    public ActionResult Details(string categoria) {
        try {
            ViewBag.Message = categoria;
            var model = (from p in db.Products
                        where p.Category.CategoryName == categoria
                        select p).ToList();
            if (model == null)
                return View("NotFound");
            else
                return View(model);
        }
        finally {
            if (db != null) db.Dispose();
        }
    }
}
}
}

```

Quando unimos e criamos um view para os exemplos anteriores, listamos todos os produtos quando clicamos em determinada categoria.

View List.aspx:

```

<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
    Inherits="System.Web.Mvc.ViewPage<IEnumerable<AppCapitulo4.Models.Category>>" %>
<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
    Lista de categorias
</asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
    <% foreach (var item in Model) { %>
        <%: Html.ActionLink(item.CategoryName, "Details",
            new { categoria = item.CategoryName })%><br />
    <% } %>
</asp:Content>

```

View Details.aspx:

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
    Inherits="System.Web.Mvc.ViewPage<IEnumerable<AppCapitulo4.Models.Product>" %>
<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
    <%= ViewBag.Message%>
</asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
    <h2><%= ViewBag.Message%></h2>
    <ul>
        <% foreach (var item in Model) { %>
            <li>
                <%= item.ProductName %>
            </li>
        <% } %>
    </ul>
    <p><%= Html.ActionLink("Volta para página anterior", "List") %></p>
</asp:Content>
```

Observe a figura 4.9:



Figura 4.9 – Exibindo categorias e produtos.

4.3.3 Adicionando um novo registro

Use o objeto Entity Framework NorthwindBD para inserir um novo registro no banco de dados:

```
db.AddToCategories(p);
db.SaveChanges();
```

Exemplo:

```
private NorthwindBD db = new NorthwindBD();
public ActionResult Create() {
    return View();
}
[HttpPost]
public ActionResult Create([Bind(Exclude = "CategoryID")]Category p) {
    try {
        if (!ModelState.IsValid) return View();
        db.AddToCategories(p);
        db.SaveChanges();
        return RedirectToAction("Index");
    }
    catch (System.Data.EntitySqlException) {
        ModelState.AddModelError("", "Erro EntitySqlException.");
        return View();
    }
    catch (System.Data.EntityCommandExecutionException) {
        ModelState.AddModelError("", "Erro EntityCommandExecutionException.");
        return View();
    }
    finally {
        if (db != null) db.Dispose();
    }
}
```

Obs.: em uma aplicação real, profissional, é necessário validar a entrada dos usuários. Use `ModelState.IsValid` e atributos específicos. Abordamos a validação de dados no capítulo 8.

4.3.4 Editando registros

Para editar um ou mais registros, usamos filtros:

```
var model = db.Categories.Where(c => c.CategoryID == id).FirstOrDefault();
```

e dois métodos `Edit`. O primeiro preenche o formulário com informações atuais do registro:

```
public ActionResult Edit(int? id) {
    try {
        if (!id.HasValue) return View("NotFound");
        var model = db.Categories.Where(c => c.CategoryID == id).FirstOrDefault();
```

```

        if (model == null)
            return View("NotFound");
        else
            return View(model);
    }
    finally {
        if (db != null) db.Dispose();
    }
}

```

O segundo filtra:

```
var model = db.Categories.Where(c => c.CategoryID == id).FirstOrDefault();
```

E envia as atualizações ao banco de dados:

```
this.TryUpdateModel(categoria);
db.SaveChanges();
```

Exemplo:

```

[HttpPost]
public ActionResult Edit(int? id, FormCollection collection) {
    try {
        if (!id.HasValue) return View("NotFound");
        var model = db.Categories.Where(c => c.CategoryID == id).FirstOrDefault();
        if (model == null) {
            return View("NotFound");
        }
        else {
            this.TryUpdateModel(model);
            db.SaveChanges();
        }
        return RedirectToAction("Index");
    }
    catch (System.Data.EntitySqlException) {
        ModelState.AddModelError("", "Erro EntitySqlException.");
        return View();
    }
    catch (System.Data.EntityCommandExecutionException) {
        ModelState.AddModelError("", "Erro EntityCommandExecutionException.");
        return View();
    }
    finally {
        if (db != null) db.Dispose();
    }
}

```

O view `NotFound` deve ser criado manualmente no diretório `Views\Shared`, e é exibido quando a busca não retorna nenhum registro.

```

<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
    Inherits="System.Web.Mvc.ViewPage<dynamic>" %>
<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
    Registro não encontrado!

```



```

</asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
    <h2>Registro não encontrado!</h2>
</asp:Content>

```

4.3.5 Excluindo registros

Excluir um registro é simples. Crie um método `Delete`:

```
public ActionResult Delete(int? id) {}
```

Identifique o registro com um filtro:

```
var model = db.Categories.Where(c => c.CategoryID == id).FirstOrDefault();
```

E invoque o método `DeleteObject`:

```
db.DeleteObject(model);
```

E o método `SaveChanges`:

```
db.SaveChanges();
```

Exemplo:

```

public ActionResult Delete(int? id) {
    try {
        if (!id.HasValue)
            return View("NotFound");
        var model = db.Categories.Where(c => c.CategoryID == id).FirstOrDefault();
        if (model == null) {
            return View("NotFound");
        }
        else {
            db.DeleteObject(model);
            db.SaveChanges();
        }
        return RedirectToAction("Index");
    }
    catch (System.Data.EntitySqlException) {
        ModelState.AddModelError("", "Erro EntitySqlException.");
        return View();
    }
    catch (System.Data.EntityCommandExecutionException) {
        ModelState.AddModelError("", "Erro EntityCommandExecutionException.");
        return View();
    }
    finally {
        if (db != null) db.Dispose();
    }
}

```

4.4 Camada de dados

Acessar um objeto Entity Framework diretamente no controlador não é a melhor opção em aplicações profissionais, pois dificulta a manutenção, os testes e duplica códigos LINQ, por exemplo.

A melhor opção é você criar uma classe e uma interface. Por exemplo, criamos dois objetos: *ICategoriesRepository*, *CategoriesRepository*. Essa técnica é denominada Injeção de dependências.

4.4.1 Interface *ICategoriesRepository*

ICategoriesRepository – Descreve os métodos usados pela aplicação.

Acrescente ao diretório *Models* da aplicação ASP.NET MVC o arquivo *ICategoryRepository.cs* com o seguinte código:

```
using System;
using System.Collections.Generic;
namespace AppCapítulo4.Models {
    public interface ICategoryRepository : IDisposable {
        IEnumerable<Category> GetAllCategory();
        Category GetCategory(int? id);
        void Create(Category entidade);
        void Edit(Category entidade);
        void Delete(Category entidade);
    }
}
```

4.4.2 Classe *CategoriesRepository*

CategoriesRepository – Implementa os métodos definidos na interface *ICategoriesRepository*.

Adicione ao diretório *Models* o arquivo *CategoryRepository.cs* com o seguinte código:

```
using System;
using System.Collections.Generic;
using System.Linq;
namespace AppCapítulo4.Models {
    public class CategoryRepository : ICategoryRepository {
        private bool disposed = false;
        private NorthwindBD db = new NorthwindBD();

        public IEnumerable<Category> GetAllCategory() {
            var categorias = db.Categories.AsParallel().ToList();
            return categorias;
        }
    }
}
```

```

public Category GetCategory(int? id) {
    var categoria = db.Categories.Where(c => c.CategoryID == id).FirstOrDefault();
    return categoria;
}

public void Create(Category entidad) {
    if (entidad != null) {
        db.AddToCategories(entidad);
        db.SaveChanges();
    }
}

public void Edit(Category entidad) {
    if (entidad != null) {
        var id = GetCategory(entidad.CategoryID);
        if (id != null) {
            db.ApplyCurrentValues(id.EntityKey.EntitySetName, entidad);
            db.SaveChanges();
        }
    }
}

public void Delete(Category entidad) {
    if (entidad != null) {
        var id = GetCategory(entidad.CategoryID);
        if (id != null) {
            db.DeleteObject(id);
            db.SaveChanges();
        }
    }
}

public void Dispose() {
    Dispose(true);
    GC.SuppressFinalize(this);
}

~CategoryRepository() {
    Dispose(false);
}

protected virtual void Dispose(bool disposing) {
    if (!this.disposed) {
        if (disposing) {
            if (db != null) {
                db.Dispose();
                db = null;
            }
        }
    }
}

```

```

        }
        disposed = true;
    }
}

```

Obs.: após a criação da interface `ICategoryRepository` e da classe `CategoryRepository`, compile a aplicação conforme a figura 4.1.

Repare que a interface `ICategoryRepository` é implementada por meio de herança:

```
public class CategoryRepository : ICategoryRepository {}
```

E uma instância da classe Entity Framework `NorthwindBD` é declarada:

```
private NorthwindBD db = new NorthwindBD();
```

4.4.2.1 Listar categorias

`GetAllCategory` lista todas as categorias:

```
public IEnumerable<Category> GetAllCategory() {
    var categorias = db.Categories.AsParallel().ToList();
    return categorias;
}

```

4.4.2.2 Retornar um único registro

`GetCategory` filtra as categorias usando o identificador:

```
public Category GetCategory(int? id) {
    var categoria = db.Categories.Where(c => c.CategoryID == id).FirstOrDefault();
    return categoria;
}

```

4.4.2.3 Inserir registros

O método `Create` insere uma nova categoria na tabela `Categories` do banco de dados `Northwind`:

```
public void Create(Category entidade) {
    if (entidade != null) {
        db.AddToCategories(entidade);
        db.SaveChanges();
    }
}

```

4.4.2.4 Editar registros

O método `Edit` edita uma categoria específica:

```

public void Edit(Category entidade) {
    if (entidade != null) {
        var id = GetCategory(entidade.CategoryID);
        if (id != null) {
            db.ApplyCurrentValues(id.EntityKey.EntitySetName, entidade);
            db.SaveChanges();
        }
    }
}

```

4.4.2.5 Excluir

O método Delete exclui uma categoria específica:

```

public void Delete(Category entidade) {
    if (entidade != null) {
        var id = GetCategory(entidade.CategoryID);
        if (id != null) {
            db.DeleteObject(id);
            db.SaveChanges();
        }
    }
}

```

4.4.2.6 Liberando recursos

Liberamos os recursos usados pela classe `CategoryRepository` por intermédio dos métodos listados a seguir:

```

public void Dispose() {
    Dispose(true);
    GC.SuppressFinalize(this);
}

~CategoryRepository() {
    Dispose(false);
}

protected virtual void Dispose(bool disposing) {
    if (!this.disposed) {
        if (disposing) {
            if (db != null) {
                db.Dispose();
                db = null;
            }
        }
    }
    disposed = true;
}

```

4.4.3 Invoque os métodos da interface

Na classe controladora, invoque os métodos da interface. Assim, qualquer alteração no código dos métodos da classe `CategoriesRepository` não afeta o código que invoca a interface. Exemplo:

```
using System.Web.Mvc;
using AppCapitulo4.Models;
namespace AppCapitulo4.Controllers {
    public class HomeController : Controller {
        private ICategoryRepository _repository;
        public HomeController() : this(new CategoryRepository()) {}
        public HomeController(ICategoryRepository repository) { _repository = repository;}
        public ActionResult Index() {
            try {
                var model = _repository.GetAllCategory();
                return View(model);
            }
            finally {
                if (_repository != null) _repository.Dispose();
            }
        }
    }
}
```

A classe controladora `HomeController` tem dois métodos construtores. O primeiro invoca o outro por meio da palavra-chave `this` e passa uma instância da classe `CategoryRepository`. Técnica utilizada quando construtores têm algum código em comum:

```
public class HomeController : Controller {
    private ICategoryRepository _repository;
    public HomeController() : this(new CategoryRepository()) {}
    public HomeController(ICategoryRepository repository) { _repository = repository; }
}
```

Invoque os métodos por intermédio da variável `_repository`:

```
var model = _repository.GetAllCategory();
```

O método `Dispose` libera os recursos utilizados pela classe `CategoryRepository`:

```
_repository.Dispose();
```

Para acrescentar uma nova categoria, use:

```
_repository.Create(c);
```

Exemplo:

```
public ActionResult Create() {
    return View();
}
```

```
[HttpPost]
public ActionResult Create(Category c) {
    try {
        if (!ModelState.IsValid)
            return View();
        _repository.Create(c);
        return RedirectToAction("Index");
    }
    catch (System.Data.EntitySqlException) {
        ModelState.AddModelError("", "Erro EntitySqlException.");
        return View();
    }
    catch (System.Data.EntityCommandExecutionException) {
        ModelState.AddModelError("", "Erro EntityCommandExecutionException.");
        return View();
    }
    finally {
        if (_repository != null) _repository.Dispose();
    }
}
```

Se você pretende atualizar determinado registro da tabela *Categories*, identifique o registro no primeiro método *Edit*:

```
public ActionResult Edit(int id) {
    try {
        var model = _repository.GetCategory(id);
        if (model == null)
            return View("NotFound");
        else
            return View(model);
    }
    finally {
        if (_repository != null) _repository.Dispose();
    }
}
```

E atualize no segundo método *Edit*:

```
[HttpPost]
public ActionResult Edit(Category c) {
    try {
        if (!ModelState.IsValid) return View();
        _repository.Edit(c);
        return RedirectToAction("Index");
    }
}
```

```

        catch(System.Data.EntitySqlException) {
            ModelState.AddModelError("", "Erro EntitySqlException.");
            return View();
        }
        catch (System.Data.EntityCommandExecutionException) {
            ModelState.AddModelError("", "Erro EntityCommandExecutionException.");
            return View();
        }
        finally {
            if (_repository != null) _repository.Dispose();
        }
    }
}

```

Procedimento semelhante adotamos quando excluimos registros:

```

public ActionResult Delete(int id) {
    try {
        var model = _repository.GetCategory(id);
        if (model == null)
            return View("NotFound");
        else
            _repository.Delete(model);
        return RedirectToAction("Index");
    }
    finally {
        if (_repository != null) _repository.Dispose();
    }
}

```

4.5 Repositório genérico

O código descrito no tópico anterior é trilegal, como se diz aqui no Rio Grande do Sul. Mas tem um problema. Note que é preciso uma interface e uma classe repositório para cada tabela do banco de dados. Em uma aplicação grande, significa muito código repetido, entre outros problemas.

A solução é usar o recurso generics do .NET Framework e desenvolver uma interface que pode ser usada por todas as tabelas do banco de dados.

Obs.: o repositório genérico exemplificado neste tópico é compatível com o ADO.NET Entity Framework 4.0. Para testá-lo baixe os arquivos de exemplo e abra no Visual Studio o projeto do capítulo 4. Nos arquivos de exemplo deste livro você encontra também um repositório genérico compatível com o ADO.NET Entity Framework 5.0 e 6.0.

4.5.1 A interface `IGenericRepository`

Neste livro, usamos a interface `IGenericRepository` para descrever os métodos usados pelas classes repositório.

Acrescente ao diretório `Models` o arquivo `IGenericRepository.cs` com o código a seguir:

```
using System;
using System.Collections.Generic;
using System.Data.Objects.DataClasses;
using System.Linq.Expressions;

namespace AppCapítulo4.Models {
    public interface IGenericRepository : IDisposable {
        IEnumerable<T> GetAll<T>();
        IEnumerable<T> Find<T>(Expression<Func<T, bool>> predicate);
        T GetSingle<T>(Expression<Func<T, bool>> predicate);
        void Create<T>(T entidade);
        void Edit<T>(T entidade, Expression<Func<T, bool>> predicate) where T : class, IEntityWithKey;
        void Delete<T>(Expression<Func<T, bool>> predicate);
    }
}
```

O método `GetAll` retorna todos os registros da entidade `T`:

```
IEnumerable<T> GetAll<T>();
```

Obs.: `T` é o nome da entidade, exemplo `category`.

A busca por registros específicos é realizada pelo método `Find` com o auxílio de expressões `lambda`:

```
IEnumerable<T> Find<T>(Expression<Func<T, bool>> predicate);
```

O método `GetSingle` retorna a primeira ocorrência de um registro restringido por expressões `lambda`:

```
T GetSingle<T>(Expression<Func<T, bool>> predicate);
```

Para operações de inserção, usamos o método `Create`:

```
void Create<T>(T entidade);
```

Operações de exclusão usam o método `Delete` e expressões `lambda`:

```
void Delete<T>(Expression<Func<T, bool>> predicate);
```

A atualização de registros é realizada pelo método `Edit`:

```
void Edit<T>(T entidade, Expression<Func<T, bool>> predicate) where T : class, IEntityWithKey;
```

Além do nome da entidade, passamos ao método `Edit` expressões `lambda` e informamos que o tipo `T` deve ser um tipo de referência e que requer a implementação da interface `IEntityWithKey`.

4.5.2 Classe EntityFrameworkRepository

A classe `EntityFrameworkRepository` implementa os métodos da interface `IGenericRepository`.

Acrescente ao diretório `Models` o arquivo `EntityFrameworkRepository.cs`. Em seguida, adicione o seguinte código ao arquivo:

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Data.Objects;
using System.Data.Objects.DataClasses;
using System.Linq;
using System.Linq.Expressions;

namespace AppCapitulo4.Models {
    public class EntityFrameworkRepository : IGenericRepository {
        private ObjectContext _contexto;
        private bool disposed = false;

        public EntityFrameworkRepository(ObjectContext dataContext) {
            _contexto = dataContext;
        }

        public IEnumerable<T> GetAll<T>() {
            string entityName = GetEntidade<T>();
            IList<T> query = _contexto.CreateQuery<T>(entityName).ToList();
            return query;
        }

        public IEnumerable<T> Find<T>(Expression<Func<T, bool>> predicate) {
            string entityName = GetEntidade<T>();
            var model = _contexto.CreateQuery<T>(entityName).Where(predicate).ToList();
            return model;
        }

        public T GetSingle<T>(Expression<Func<T, bool>> predicate) {
            string entityName = GetEntidade<T>();
            var model = _contexto.CreateQuery<T>(entityName).Where(predicate).FirstOrDefault<T>();
            return model;
        }

        public void Create<T>(T entidade) {
            if (entidade != null) {
                string entityName = GetEntidade<T>();
                _contexto.AddObject(entityName, entidade);
                _contexto.SaveChanges();
            }
        }
    }
}
```

```

public void Edit<T>(T entidade, Expression<Func<T, bool>> predicate) where T : class,
    IEntityWithKey {
    if (entidade != null) {
        string entityName = GetEntidade<T>();
        var model = GetSingle(predicate);
        if (model != null) {
            _contexto.ApplyCurrentValues(model.EntityKey.EntitySetName, entidade);
            _contexto.SaveChanges();
        }
    }
}

public void Delete<T>(Expression<Func<T, bool>> predicate) {
    var model = GetSingle(predicate);
    if (model != null) {
        _contexto.DeleteObject(model);
        _contexto.SaveChanges();
    }
}

private string GetEntidade<T>() {
    var EntitySetName = _contexto.GetType().GetProperties()
        .Single(p => p.PropertyType.IsGenericType && typeof(IEnumerable<>)
            .MakeGenericType(typeof(T)).IsAssignableFrom(p.PropertyType));
    return EntitySetName.Name;
}

public void Dispose() {
    Dispose(true);
    GC.SuppressFinalize(this);
}

~EntityFrameworkRepository() {
    Dispose(false);
}

protected virtual void Dispose(bool disposing) {
    if (!this.disposed) {
        if (disposing) {
            if (_contexto != null) {
                _contexto.Dispose();
                _contexto = null;
            }
        }
    }
    disposed = true;
}
}
}

```

Obs.: baixe os arquivos de exemplo no website da Editora Novatec e copie o arquivo `EntityFrameworkRepository.cs` para o diretório `Models`, assim você não precisa digitar grandes quantidades de código. Compile a aplicação conforme a figura 4.1.

A classe `EntityFrameworkRepository` implementa por meio de herança as funções da interface `IGenericRepository`:

```
public class EntityFrameworkRepository : IGenericRepository {}
```

E contém um método construtor

```
public class EntityFrameworkRepository : IGenericRepository {
    private ObjectContext _contexto;
    private bool disposed = false;
    public EntityFrameworkRepository(ObjectContext dataContext) {
        _contexto = dataContext;
    }
}
```

Cujo parâmetro aceita uma classe do ADO.NET Entity Framework herdada de `ObjectContext`.

Exemplo:

```
_repository = new EntityFrameworkRepository(new NorthwindBD());
```

4.5.2.1 Método GetAll

O método `GetAll` extrai todas as linhas de qualquer tabela. Para que isso seja possível, é necessária a criação de uma consulta dinâmica, com o método `CreateQuery`, que se adapte a qualquer tabela:

```
ICollection<T> query = _contexto.CreateQuery<T>(entityName).ToList();
```

Exemplo:

```
public IEnumerable<T> GetAll<T>() {
    string entityName = GetEntidade<T>();
    ICollection<T> query = _contexto.CreateQuery<T>(entityName).ToList();
    return query;
}
```

O método `GetEntidade<T>` retorna o valor da propriedade `EntitySetName` de uma entidade:

```
private string GetEntidade<T>() {
    var EntitySetName = _contexto.GetType().GetProperties()
        .Single(p => p.PropertyType.IsGenericType && typeof(IEnumerable<>)
            .MakeGenericType(typeof(T)).IsAssignableFrom(p.PropertyType));
    return EntitySetName.Name;
}
```

4.5.2.2 Método Find

O método Find encontra todas as ocorrências de acordo com os critérios definidos na expressão lambda:

```
public IEnumerable<T> Find<T>(Expression<Func<T, bool>> predicate) {
    string entityName = GetEntidade<T>();
    var model = _contexto.CreateQuery<T>(entityName).Where(predicate).ToList();
    return model;
}
```

4.5.2.3 Método GetSingle

Retorna a primeira ocorrência de um registro de acordo com os critérios especificados pela expressão lambda:

```
public T GetSingle<T>(Expression<Func<T, bool>> predicate) {
    string entityName = GetEntidade<T>();
    var model = _contexto.CreateQuery<T>(entityName).Where(predicate).FirstOrDefault<T>();
    return model;
}
```

4.5.2.4 Método Create

O método Create insere um novo registro na entidade passada ao método:

```
public void Create<T>(T entidade) {
    if (entidade != null) {
        string entityName = GetEntidade<T>();
        _contexto.AddObject(entityName, entidade);
        _contexto.SaveChanges();
    }
}
```

4.5.2.5 Método Edit

O método Edit é o mais complexo de todos. Possui dois parâmetros:

```
public void Edit<T>(T entidade, Expression<Func<T, bool>> predicate)
```

e restrições quanto à implementação:

```
where T : class, IEntityWithKey
```

Usamos a expressão lambda para filtrar o registro que deve ser editado:

```
var model = GetSingle(predicate);
```

E o nome da entidade é usado para aplicar as alterações:

```
_contexto.ApplyCurrentValues(model.EntityKey.EntitySetName, entidade);
_contexto.SaveChanges();
```

Exemplo:

```
public void Edit<T>(T entidade, Expression<Func<T, bool>> predicate) where T : class,
    IEntityWithKey {
    if (entidade != null) {
        string entityName = GetEntidade<T>();
        var model = GetSingle(predicate);
        if (model != null) {
            _contexto.ApplyCurrentValues(model.EntityKey.EntitySetName, entidade);
            _contexto.SaveChanges();
        }
    }
}
```

Verifique os objetos antes de realizar uma operação. Por exemplo, antes de usar uma entidade, verificamos a sua existência:

```
if (entidade != null) {}
```

O mesmo ocorre quando usamos um objeto retornado por um método:

```
var model = GetSingle(predicate);
if (model != null) {}
```

4.5.2.6 Método Delete

Exclui um registro específico definido na expressão lambda:

```
public void Delete<T>(Expression<Func<T, bool>> predicate) {
    var model = GetSingle(predicate);
    if (model != null) {
        _contexto.DeleteObject(model);
        _contexto.SaveChanges();
    }
}
```

4.5.2.7 Método Dispose

O método Dispose libera os recursos utilizados pela classe EntityFrameworkRepository:

```
public void Dispose() {
    Dispose(true);
    GC.SuppressFinalize(this);
}

~EntityFrameworkRepository() {
    Dispose(false);
}
```

```
protected virtual void Dispose(bool disposing) {
    if (!this.disposed) {
        if (disposing) {
            if (_contexto != null) {
                _contexto.Dispose();
                _contexto = null;
            }
        }
    }
    disposed = true;
}
```

4.5.3 Invocando os métodos da interface generics

Para invocar os métodos da interface `IGenericRepository`, crie no método construtor do controlador uma nova instância da classe repositório `EntityFrameworkRepository` e passe ao método uma instância da classe ADO.NET Entity Framework `NorthwindBD` herdada de `ObjectContext`:

```
public class HomeController : Controller {
    private IGenericRepository _repository;
    public HomeController() {
        _repository = new EntityFrameworkRepository(new NorthwindBD());
    }
}
```

4.5.3.1 Exibir todos os registros

Para exibir todos os registros de determinada tabela, use o método `GetAll<Entidade>()`.

Exemplo:

```
public ActionResult Index() {
    try {
        var model = _repository.GetAll<Category>();
        return View(model);
    }
    finally {
        if (_repository != null) _repository.Dispose();
    }
}
```

Para exibir os registros de outra tabela, basta alterar o nome da entidade. Exemplo:

```
var model = _repository.GetAll<Customer>();
var model1 = _repository.GetAll<Product>();
var model2 = _repository.GetAll<Order>();
```

4.5.3.2 Detalhes de um registro

Para exibir os detalhes de um registro específico, use o método `GetSingle`:

```
public ActionResult Single(int? id) {
    try {
        if (!id.HasValue)
            return View("NotFound");
        var model = _repository.GetSingle<Category>(c => c.CategoryID == id);
        if (model == null)
            return View("NotFound");
        else
            return View(model);
    }
    finally {
        if (_repository != null) _repository.Dispose();
    }
}
```

O método `GetSingle` requer o nome da entidade e também a expressão lambda:

```
var model = _repository.GetSingle<Category>(c => c.CategoryID == id);
```

4.5.3.3 Busca

A busca por informações específicas no banco de dados pode ser realizada pelo método `Find`. Exemplo:

```
public ActionResult Details(string categoria) {
    try {
        ViewBag.Message = categoria;
        var model = _repository.Find<Product>(p => p.Category.CategoryName == categoria).
            ToList();
        return View(model);
    }
    finally {
        if (_repository != null) _repository.Dispose();
    }
}
```

No exemplo, exibimos todos os produtos de determinada categoria.

4.5.3.4 Inserindo registros

O primeiro método `Create` exibe o formulário em branco:

```
public ActionResult Create() {
    return View();
}
```

Obs.: para exibir o formulário, é preciso associar um view ao método `Create`.

No segundo método `Create` definimos o nome da entidade e o argumento passado:

```
_repository.Create<Category>(categoria);
```

e enviamos as informações ao banco de dados. Exemplo:

```
[HttpPost]
public ActionResult Create(Category categoria) {
    try {
        if (!ModelState.IsValid)
            return View();
        _repository.Create<Category>(categoria);
        return RedirectToAction("Index");
    }
    catch {
        return View();
    }
    finally {
        if (_repository != null) _repository.Dispose();
    }
}
```

4.5.3.5 Editando registros

Quando editamos informações no banco de dados, definimos qual registro deve ser editado no primeiro método `Edit` do controlador:

```
public ActionResult Edit(int? id) {
    try {
        if (!id.HasValue)
            return View("NotFound");
        var model = _repository.GetSingle<Category>(c => c.CategoryID == id);
        if (model == null)
            return View("NotFound");
        else
            return View(model);
    }
    finally {
        if (_repository != null) _repository.Dispose();
    }
}
```

No segundo método `Edit` inserimos as informações no banco de dados:

```
[HttpPost]
public ActionResult Edit(Category categoria) {
    try {
        _repository.Edit<Category>(categoria, p => p.CategoryID == categoria.CategoryID);
    }
}
```

```
        return RedirectToAction("Index");
    }
    catch {
        return View();
    }
    finally {
        if (_repository != null) _repository.Dispose();
    }
}
```

4.5.3.6 Excluindo registros

Para excluir um registro, usamos dois métodos `Delete`. Um seleciona o registro que deve ser excluído e o segundo método contém o código que exclui o registro do banco de dados.

Exemplo:

```
public ActionResult Delete(int? id) {
    try {
        if (!id.HasValue)
            return View("NotFound");

        _repository.Delete<Category>(p => p.CategoryID == id);
        return RedirectToAction("Index");
    }
    finally {
        if (_repository != null) _repository.Dispose();
    }
}
```

Razor

A sintaxe Razor embute em uma mesma página web código que roda no servidor, marcações HTML, seletores CSS e código de linguagens de script, como JavaScript.

A sintaxe Razor é simples, fácil de digitar e, o melhor, você não precisa aprender uma nova linguagem de programação.

O caractere `@` é o cara no Razor. É usado para declarar blocos de instruções, retornar o resultado de expressões, invocar métodos, acessar propriedades, declarar variáveis, criar uma instância de uma classe etc.

A sintaxe do Razor é extremamente simples. Por exemplo, para retornar a data atual, usamos a linha a seguir:

```
<p>@DateTime.Now</p>
```

As instruções são iniciadas com o caractere `@` e cercadas por chaves:

```
@{  
    var data = DateTime.Now;  
}
```

E podem conter múltiplas linhas de código:

```
@{  
    string nome = "Alfredo Lotar";  
    var data = DateTime.Now;  
}
```

todas finalizadas com ponto e vírgula.

5.1 Variáveis

Você pode declarar variáveis normalmente definindo o tipo de dados:

```
@{  
    string nome = "Alfredo Lotar";  
    string[] cores = new string[3];  
    int x = 10;
```

```

        int y = 50;
        int resultado = x + y;
    }
<!DOCTYPE html>
<html>
    <body>
        <div>
            <p>Total: @resultado</p>
            <p>Multiplicação: @(x*y)</p>
        </div>
    </body>
</html>

```

Ou usando a palavra-chave var:

```

@{
    var nome = "Alfredo Lotar";
    var cores = new string[3];
    var x = 10;
    var y = 50;
    var resultado = x + y;
}
<p>Total: @resultado</p>

```

Uma expressão pode ser colocada dentro de parênteses:

```
<p>@(x * y)</p>
```

O caractere @ pode ser usado também para impedir que determinados caracteres sejam processados. Por exemplo, ao armazenar o caminho de um arquivo como string, o conteúdo da variável deve ser precedido do caractere @:

```
string caminho = @"c:\livro\capítulo5";
```

O mesmo ocorre quando incluímos aspas duplas em uma string:

```
string nome = @"Alfredo ""Lotar""";
```

Ou quando dividimos o conteúdo de uma variável do tipo string em várias linhas:

```

string nome = @"Alfredo Lotar
    Programador
    autor";

```

Obs.: a sintaxe Razor quando usada com C# faz distinção entre letras maiúsculas e minúsculas.

5.2 Comentários

No Razor, comentários começam com o `@*` e terminam com `*@`. Isso se aplica a comentários em múltiplas linhas ou em linha única.

```
@{  
    @*Este comentário tem uma única linha*@  
  
    @*  
        Comentário dividido em  
        várias linhas  
    *@  
}
```

Se preferir, use os caracteres de comentário do C#:

```
@{  
    // Este comentário tem uma única linha  
  
    /*  
        Comentário dividido em  
        várias linhas  
    */  
}
```

As marcações HTML são comentadas com `<!--` e `-->`.

Exemplo:

```
<!DOCTYPE html>  
<html>  
    <body>  
        <div>  
            <p>  
                <!-- Comentário dentro de uma página com marcações HTML -->  
            </p>  
        </div>  
    </body>  
</html>
```

5.3 Instruções if

A instrução `if` retorna verdadeiro ou falso com base em um teste específico:

```
@{  
    int ano = DateTime.Now.Year;  
    if (ano > 2012) {  
        <p>Ano: @DateTime.Now.Year </p>  
    }  
}
```

```

else if (ano < 2012) {
    <p>Ano: 2011</p>
}
else {
    <p>Ano indefinido.</p>
}
}

```

Ao observar o código, notamos que é possível incluir marcações HTML no código C#:

```

if (ano > 2012) {
    <p>Ano: @DateTime.Now.Year </p>
}

```

Texto puro sem marcações HTML pode ser exibido na tela quando precedido de @:

Exemplo:

```

@{
    @:Isto é um texto sem marcações HTML
}

```

ou da marcação <text>:

```

@{
    <text>Isto é um texto sem marcações HTML</text>
}

```

5.4 Instrução switch

A instrução switch testa se determinado valor combina com uma instrução case. Se retorna verdadeiro, o código da instrução case é executado.

Exemplo:

```

@{
    int mes = DateTime.Now.Month;
    string mensagem = "";

    switch (mes) {
        case 1:
            mensagem = "Janeiro";
            break;
        case 2:
            mensagem = "Fevereiro";
            break;
        case 3:
            mensagem = "Março";
            break;
    }
}

```

```

        case 4:
            mensagem = "Abril";
            break;
        case 5:
            mensagem = "Maio";
            break;
        case 6:
            mensagem = "Junho";
            break;
        default:
            <p>Mês indefinido.</p>
            break;
    }
}
<!DOCTYPE html>
<html>
    <body>
        <div>
            <p>
                @mensagem
            </p>
        </div>
    </body>
</html>

```

5.5 Instrução for

A instrução `for` é caracterizada como loop contador, ou seja, você percorre os extremos conhecidos:

```

@{
    for (int i = 0; i < 10; i++) {
        <p>Número: @i</p>
    }
}

```

5.6 Instrução foreach

A instrução `foreach` é usada para percorrer itens de uma coleção:

```

@{
    string[] cidades = new string[3];
    cidades[0] = "São Paulo";
    cidades[1] = "Rio de Janeiro";
    cidades[2] = "Porto Alegre";
}

```

```

    <ul>
        @foreach (var cidade in cidades) {
            <li>@cidade</li>
        }
    }
</ul>
}

```

5.7 Propriedades e métodos

Com o caractere @ acessamos propriedades:

```

@{
    @Request.UserLanguages[0] <br/>
    @Request.Url <br/>
    @Request.UserAgent
}

```

e invocamos Métodos:

```

@{Html.RenderAction("ListaProdutos");}
@{
    @Server.HtmlEncode("<b>tags HTML desativadas.</b>");
}

```

5.8 Manipular exceções

Para manipular exceções com Razor, basta incluir um bloco try e catch.

Exemplo:

```

@{
    try {
        var fs = File.Open(@"c:\teste100.txt", FileMode.Open);
    }
    catch (FileNotFoundException) {
        <p>Arquivo não encontrado.</p>;
    }
    catch (Exception) {
        <p>Erro detectado.</p>;
    }
}

```


5.9 Conversões

A sintaxe Razor possui diversos métodos que nos permitem forçar uma conversão explícita de um tipo de dados para outro.

Além dos métodos usados na conversão, há métodos que permitem testes em dados. Por exemplo, podemos verificar se determinado tipo é um inteiro antes de convertê-lo.

A seguir, listamos os métodos usados na conversão e verificação de dados.

Método	Descrição
AsInt(), IsInt()	AsInt converte uma string para um inteiro. IsInt verifica se é um tipo int válido.
AsBool(), IsBool()	AsBool converte uma string para um tipo booleano (true, false). IsBool verifica se é um tipo bool válido.
AsFloat(), IsFloat()	AsFloat converte uma string para um tipo ponto flutuante. IsFloat verifica se é um tipo float válido.
AsDecimal(), IsDecimal()	AsDecimal converte uma string para um tipo decimal. IsDecimal verifica se é um tipo decimal válido.
AsDateTime(), IsDateTime()	AsDateTime converte uma string para um tipo data e hora. IsDateTime verifica se é um tipo DateTime válido.
ToString()	Converte qualquer tipo de dados em um tipo string.

Exemplo:

```
@{
    string numero = "123456";
    int i = 0;

    if (numero.IsInt()) {
        i = numero.AsInt();
    }
    <p>@i</p>
}
```

5.10 Operadores

Os operadores do C# podem ser usados normalmente com a sintaxe Razor.

Exemplo:

```
@{
    int x = 12;
    int y = 5;
    int resultado = 0;
```

```

    if (x >= 10 && y < 7) {
        resultado = y + y;
    }
    else {
        resultado = y - y;
    }
}
<!DOCTYPE html>
<html>
    <head></head>
    <body>
        <div>
            <p>
                @resultado
            </p>
        </div>
    </body>
</html>

```

5.11 Href

O método `Href` converte endereços relativos de uma imagem, arquivo CSS, página web para um endereço que o navegador entende.

Exemplo:

```


<link rel="stylesheet" type="text/css" href="@Href("~/Content/Site.css")" />

```

O mecanismo de exibição ASPX usa o método `ResolveUrl`.

Exemplo:

```

<%:Html.Image("idImag", ResolveUrl("~/Content/imagem.jpg"), "Descrição")%>

```

5.12 Associando classes

A sintaxe Razor usa a palavra-chave `model` para associar classes a um view:

```

@model IEnumerable<AppCapitulo5.Models.Paises>

```

Acessar os membros da classe associada é simples. Observe a figura 5.1:

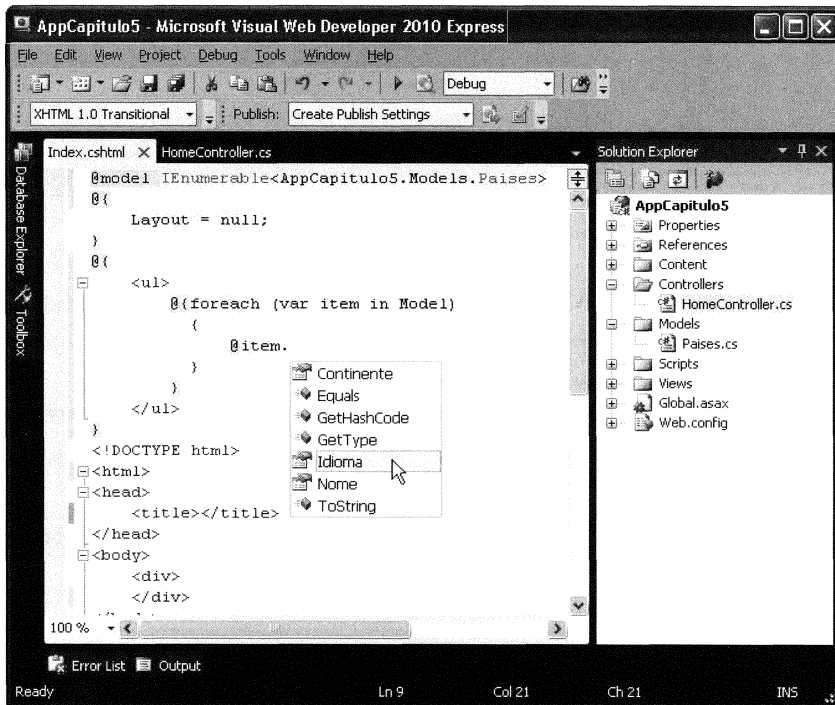


Figura 5.1 – Acessando os membros da classe Países.

5.13 Adicionando um layout

A sintaxe Razor usa layouts para definir conteúdo comum a todas as páginas de um website. É semelhante a uma master page.

Para adicionar um layout Razor a um website: na janela **Solution Explorer**, clique com o botão direito do mouse no diretório `/View/Shared`. Em seguida, selecione **Add** e clique em **New Item...**. Observe a figura 5.2.

Obs.: o diretório Shared contém os arquivos compartilhados pela aplicação. Como master pages, layouts, páginas com mensagens de erro-padrão etc.

Na caixa de diálogo **Add New Item**, selecione **MVC 4 Layout Page (Razor)**. Defina o nome do layout na caixa de texto **Name:**. Clique em **Add** para finalizar. Observe a figura 5.3.

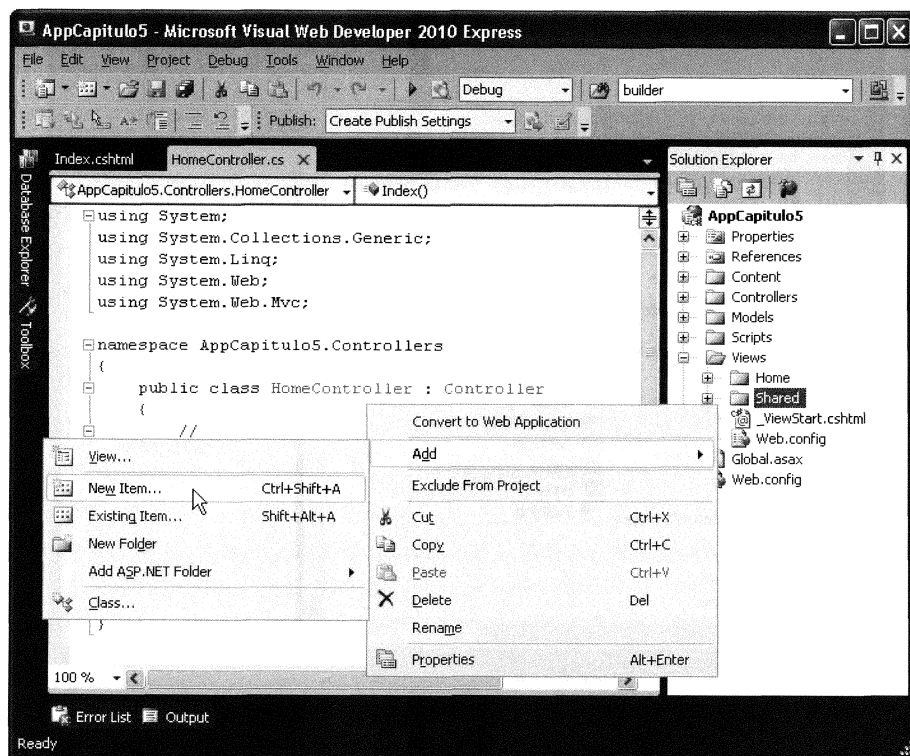


Figura 5.2 – Passos para acrescentar um layout.

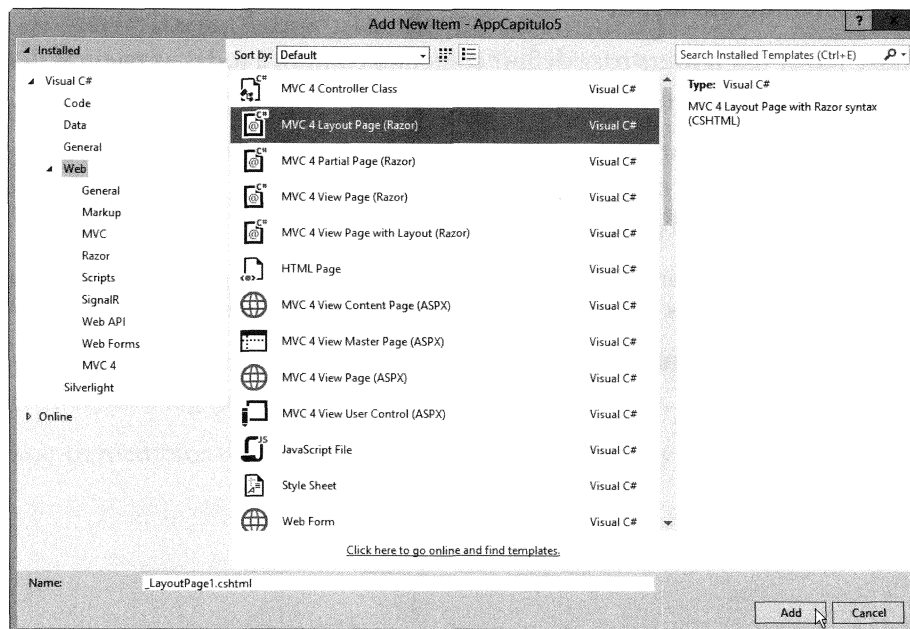


Figura 5.3 – Caixa de diálogo Add New Item.

Ao clicar em **Add**, criamos o arquivo `_LayoutPage1.cshtml` com o código a seguir:

```
<!DOCTYPE html>
<html>
  <head>
    <title>@ViewBag.Title</title>
  </head>
  <body>
    <div>
      @RenderBody()
    </div>
  </body>
</html>
```

`RenderBody()` exibe o conteúdo do view ao qual o layout está associado:

```
<div>
  @RenderBody()
</div>
```

No view a propriedade `Layout` define o nome e o caminho do layout usado:

```
@{
  ViewBag.Title = "Título da página";
  Layout = "~/Views/Shared/_LayoutPage1.cshtml";
}
<p>
  Este é o conteúdo do Website.
</p>
```

Obs.: quando usamos uma página de layout, o código HTML fica nesta. O view contém somente o conteúdo da página.

5.14 Criando um novo view Razor

Após a criação do layout, acrescentamos à aplicação um novo view Razor. Observe a figura 5.4.

Usamos o mecanismo de exibição Razor(CSHTML), a classe `Países` e o template `Details`. Marcamos a caixa de seleção **Use a layout or master page:** e definimos o nome da página de layout usada — `~/Views/Shared/_LayoutPage1.cshtml`.

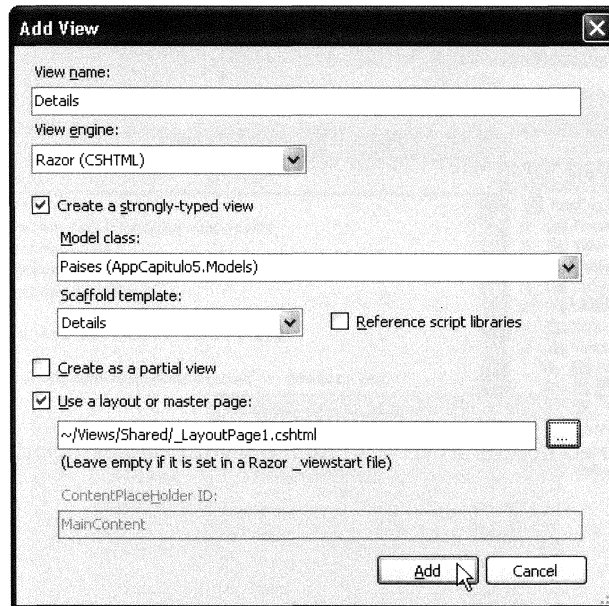


Figura 54 – Caixa de diálogo Add View.

A seguir, temos o código gerado para o view Razor:

```
@model AppCapítulo5.Models.Paises
@{
    ViewBag.Title = "Details";
    Layout = "~/Views/Shared/_LayoutPage1.cshtml";
}
<h2>Details</h2>
<fieldset>
    <legend>Países</legend>
    <div class="display-label">Nome</div>
    <div class="display-field">@Model.Nome</div>

    <div class="display-label">Continente</div>
    <div class="display-field">@Model.Continente</div>

    <div class="display-label">Idioma</div>
    <div class="display-field">@Model.Idioma</div>
</fieldset>
<p>
    @Html.ActionLink("Edit", "Edit", new { /* id=Model.PrimaryKey */ }) |
    @Html.ActionLink("Back to List", "Index")
</p>
```

5.15 Páginas com Visual consistente

Por intermédio do método `RenderPage`, incluímos o conteúdo de qualquer arquivo em uma página de layout. Imagine um arquivo que contém o conteúdo do cabeçalho:

```
<div class="cabecalho">Este é o texto do cabeçalho.</div>
```

E outro o rodapé da página:

```
<div class="rodape">Este é o texto do rodapé.</div>
```

Observe a figura 5.5:

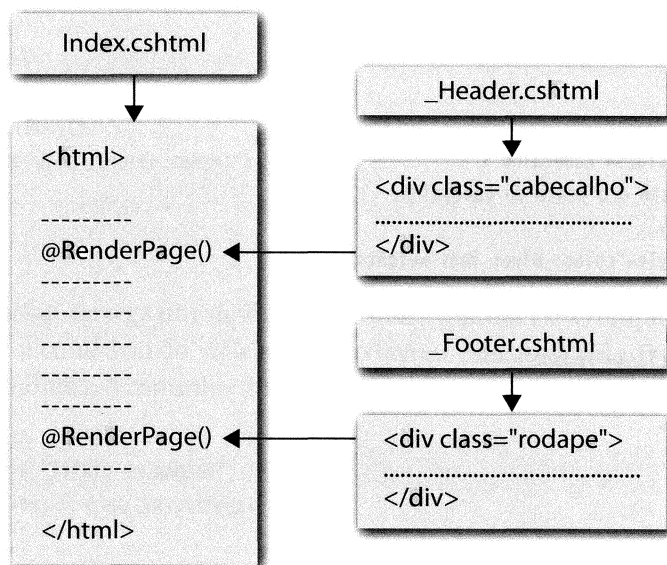


Figura 5.5 – Ilustra como uma página é inserida com o método `RenderPage`.

Eles podem ser inseridos na página de layout facilmente, exemplo:

```
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
    <head>
        <title>Título da página</title>
    </head>
    <body>
        <div>
            @RenderPage("/Views/Shared/_Header.cshtml")
            <p style="color: Blue; font-weight: bold;">
                Este é o conteúdo da página principal.
            </p>
        </div>
    </body>
</html>
```

```
        </p>
        @RenderPage("/Views/Shared/_Footer.cshtml")
    </div>
</body>
</html>
```

Quando executamos o exemplo e exibimos o código-fonte da página, temos o seguinte código:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Título da página</title>
  </head>
  <body>
    <div>
      <div class="cabecalho">
        Este é o texto do cabeçalho.
      </div>
      <p style="color: Blue; font-weight: bold;">
        Este é o conteúdo da página principal.
      </p>
      <div class="rodape">
        Este é o texto do rodapé.
      </div>
    </div>
  </body>
</html>
```

O resultado vemos na figura 5.6:

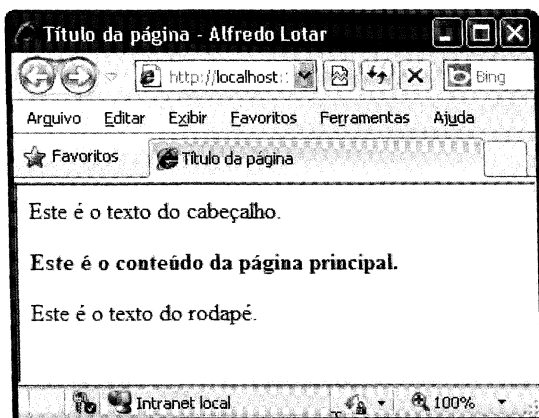


Figura 5.6 – Cabeçalho e rodapé exibidos com o método `RenderPage`.

5.15.1 Páginas de layout e o método `RenderPage`

Uma estratégia interessante é definir o conteúdo comum a todas as páginas, por exemplo, o cabeçalho e o rodapé em uma página de layout – semelhante a uma master page.

Exemplo:

```
<!DOCTYPE html>
<html>
  <head>
    <title>@ViewBag.Title</title>
  </head>
  <body>
    <div>
      @RenderPage("_Header.cshtml")
      @RenderBody()
      @RenderPage("_Footer.cshtml")
    </div>
  </body>
</html>
```

O método `RenderPage` carrega um view ou usercontrol (partial view) enquanto o método `RenderBody` exibe o conteúdo do view ao qual o layout está associado. E a propriedade `Layout` define o nome e o caminho do layout usado no view `Index.cshtml`:

```
@{
    ViewBag.Title = "Título da página";
    Layout = "~/Views/Shared/_LayoutPage1.cshtml";
}
```

O view `Index.cshtml` exibido na página de layout `_LayoutPage1.cshtml` contém o seguinte código:

```
@{
    ViewBag.Title = "Título da página";
    Layout = "~/Views/Shared/_LayoutPage1.cshtml";
}
<div>
  <p style="color: Blue; font-weight: bold;">
    Este é o conteúdo da página principal.
  </p>
</div>
```

Repare que o view contém somente o conteúdo. O restante do código HTML é inserido pela página de layout.

5.15.2 Seções

Em vez de usar vários arquivos, um para cada seção, é possível definir o conteúdo de várias seções em um único arquivo.

Exemplo:

```
@{
    ViewBag.Title = "Título da página";
    Layout = "~/Views/Shared/_LayoutPage2.cshtml";
}

@section Header {
    <div class="cabecalho">
        Este é o texto do cabeçalho.
    </div>
}
<p style="color: Blue; font-weight: bold;">
    Este é o conteúdo da página principal.
</p>
@section Footer {
    <div class="rodape">
        Este é o texto do rodapé.
    </div>
}
```

E, em seguida, exibi-las em um layout predefinido com o método `RenderSection`:

```
<!DOCTYPE html>
<html>
    <head>
        <title>@ViewBag.Title</title>
    </head>
    <body>
        <div>
            @RenderSection("Header")
            @RenderBody()
            @RenderSection("Footer")
        </div>
    </body>
</html>
```

Se a seção é opcional, defina o segundo parâmetro como `false`:

```
@RenderSection("Header", false)
```

5.16 PageData[key], PageData[index], Page

As propriedades `PageData` e `Page` contêm os dados compartilhados entre páginas, layout e partial views.

Exemplo, em um método de controlador, defina:

```
PageData["Cor"] = "Azul";
PageData[1] = "Carro";
Page.Mensagem = "Sintaxe Razor";
```

Para acessar o conteúdo das propriedades em um view e/ou layout, use:

```
<p>
    @PageData["Cor"]
    @PageData[1]
    @Page.Mensagem
</p>
```

5.17 WebGrid Helper

Apresenta as informações no formato linhas e colunas, semelhante a uma planilha do Microsoft Excel.

Para ilustrar o uso do HTML helper `WebGrid`, acrescentamos à classe `Países` o método `GetPaíses`:

```
using System;
using System.Collections.Generic;
namespace AppCapitulo5.Models {
    public class Países {
        public string Nome { get; set; }
        public string Continente { get; set; }
        public string Idioma { get; set; }

        public static List<Países> GetPaíses() {
            return new List<Países> {
                new Países { Nome="Brasil", Continente="América do sul", Idioma="Português" },
                new Países { Nome="Alemanha", Continente="Europa", Idioma="Alemão" },
                new Países { Nome="Austrália", Continente="Oceania", Idioma="Inglês" },
                new Países { Nome="Uruguai", Continente="América do sul", Idioma="Espanhol" },
                new Países { Nome="Japão", Continente="Ásia", Idioma="Japonês" }
            };
        }
    }
}
```

Em seguida, adicionamos ao controlador `HomeController` o método `List` e invocamos o método `GetPaíses` da classe `Países`:

```
using System.Web.Mvc;
using AppCapitulo5.Models;
namespace AppCapitulo5.Controllers {
    public class HomeController : Controller {
        public ActionResult Index() {
            return View();
        }
        public ActionResult List() {
            var model = Países.GetPaíses();
            return View(model);
        }
    }
}
```

Após a definição da classe Países e do método List, em um view Razor, você pode conectar a origem de dados ao WebGrid helper.

```
WebGrid grid = new WebGrid(source: Model);
```

E no topo do view importar a namespace:

```
@model IEnumerable<AppCapitulo5.Models.Países>
```

Exemplo:

```
@model IEnumerable<AppCapitulo5.Models.Países>
@{
    Layout = null;
}
@{
    WebGrid grid = new WebGrid(Model);
}
```

Ou:

```
@model IEnumerable<AppCapitulo5.Models.Países>
@{
    Layout = null;
}
@{
    WebGrid grid = new WebGrid(Model);
}
```

O parâmetro source define a origem de dados usada. Os nomes dos campos são passados por intermédio do parâmetro columnNames:

```
WebGrid grid = new WebGrid(source: Model, columnNames: new string[]
{ "Nome", "Continente", "Idioma" });
```

defaultSort define o campo usado na ordenação dos registros da grade:

```
WebGrid grid = new WebGrid(source: Model, columnNames: new string[]
{ "Nome", "Continente", "Idioma" }, defaultSort: "Nome");
```

O parâmetro `canSort` habilita ou desabilita a ordenação de registros:

```
WebGrid grid = new WebGrid(source: Model, columnNames: new string[]  
    { "Nome", "Continente", "Idioma" }, defaultSort: "Nome", canSort:true);
```

O parâmetro `canPage` ativa ou desativa a paginação:

```
WebGrid grid = new WebGrid(source: Model, columnNames: new string[]  
    { "Nome", "Continente", "Idioma" }, defaultSort: "Nome", canSort:true, canPage:true);
```

O número de registros por página é determinado por intermédio do parâmetro `rowsPerPage`:

```
WebGrid grid = new WebGrid(source: Model, columnNames: new string[] { "Nome", "Continente",  
    "Idioma" }, defaultSort: "Nome", canSort:true, canPage:true, rowsPerPage:10);
```

Se preferir, use o método `Bind` para conectar o `WebGrid` helper à origem de dados:

```
WebGrid grid = new WebGrid();  
grid.Bind(source: Model, columnNames: new string[] { "Nome", "Continente", "Idioma" });
```

Nesse caso, a paginação e a ordenação dos registros são habilitadas e desabilitadas com o parâmetro `autoSortAndPage`:

```
WebGrid grid = new WebGrid();  
grid.Bind(source: Model, columnNames: new string[]  
    { "Nome", "Continente", "Idioma" }, autoSortAndPage:true);
```

O método `GetHtml` exibe os registros do `WebGrid` helper no navegador:

```
WebGrid grid = new WebGrid(rowsPerPage:2);  
grid.Bind(source: Model, columnNames: new string[] { "Nome", "Continente", "Idioma" },  
    autoSortAndPage:true);  
@grid.GetHtml()
```

Observe a figura 5.7:

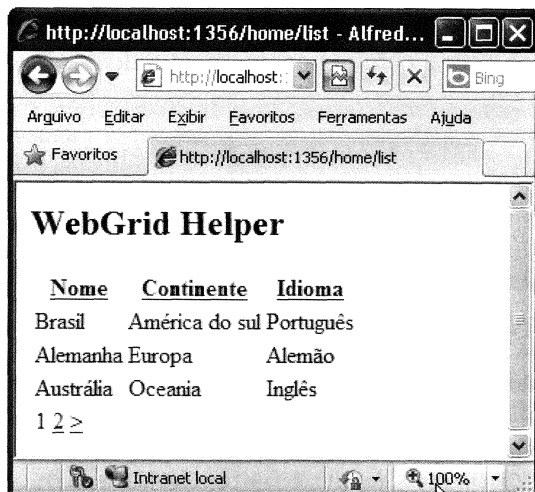


Figura 5.7 – `WebGrid` helper em ação.

5.17.1 Formatando colunas

Se você precisa formatar as colunas antes de exibi-las, use o método `Column` com o método `Columns`:

```
WebGrid grid = new WebGrid(source: Model);

@grid.GetHtml(columns: grid.Columns(
    grid.Column("Nome", header: "Nome", format: @<i>@item.Nome</i>),
    grid.Column("Continente", format: @<i>@item.Continente</i>),
    grid.Column("Idioma", format: @<text>** @item.Idioma **</text>)
    )
);
```

O nome do campo exibido é definido com o parâmetro `columnName` e `header` define o texto do cabeçalho:

```
grid.Column(columnName: "Nome", header: "Nome")
```

O parâmetro `format` aplica uma formatação específica ao conteúdo:

```
grid.Column(columnName: "Nome", header: "Nome", format: @<i>@item.Nome</i>)
```

Ou simplesmente escreve `R$` antes de um valor numérico:

```
grid.Column("Price", header: "Preço", format: @<text>R$@item.Price</text>)
```

Algo mais sofisticado também é possível.

Exemplo:

```
grid.Column("Price", header: "Preço", format: @<text>@string.Format(new System.Globalization.
    CultureInfo("pt-BR"), "{0:c}", @item.Price)</text>)
```

5.17.2 Aplicando uma folha de estilo CSS

Defina um seletor CSS para a coluna por intermédio do parâmetro `style`:

```
grid.Column("Idioma", format: @<text>** @item.Idioma **</text>, style: "SeletorCSS")
```

A aparência da tabela, do cabeçalho, do rodapé e das linhas pode ser definida com o método `GetHtml`:

```
@{
    WebGrid grid = new WebGrid(source: Model);
    @grid.GetHtml(
        tableStyle: "grid",
        headerStyle: "head",
        columns: grid.Columns(
            grid.Column("Nome", header: "Nome", format: @<i>@item.Nome</i>),
            grid.Column("Continente", format: @<i>@item.Continente</i>),
```

```

        grid.Column("Idioma", format: @<text>** @item.Idioma **</text>)
    )
}

```

O parâmetro `alternatingRowStyle` exibe uma grade com uma linha de cada cor, facilitando a identificação do conteúdo:

```

@{
    WebGrid grid = new WebGrid(source: Model);
    @grid.GetHtml(
        tableStyle: "grid",
        headerStyle: "head",
        alternatingRowStyle: "alt",
        columns: grid.Columns(
            grid.Column("Nome", header: "Nome", format: @<i>@item.Nome</i>),
            grid.Column("Continente", format: @<i>@item.Continente</i>),
            grid.Column("Idioma", format: @<text>** @item.Idioma **</text>)
        )
    )
}

```

Observe a figura 5.8:



Figura 5.8 – WebGrid helper com o parâmetro `alternatingRowStyle` definido.

Obs.: `grid`, `head` e `alt` são seletores CSS definidos no arquivo `Content/Site.css`.

5.17.3 Outros parâmetros

O parâmetro `caption` contém o título do gráfico. O parâmetro `displayHeader` ativa ou desativa-o:

```
@grid.GetHtml(
    tableStyle: "grid",
    headerStyle: "head",
    alternatingRowStyle: "alt",
    caption: "Título do gráfico",
    displayHeader: true,
    columns: grid.Columns(
        grid.Column("Nome", header: "Nome", format: @<i>@item.Nome</i>),
        grid.Column("Continente", format: @<i>@item.Continente</i>),
        grid.Column("Idioma", format: @<text>** @item.Idioma **</text>)
    )
)
```

O conteúdo alternativo exibido em células vazias é definido com o parâmetro `emptyRowCellValue` e é ativado com o parâmetro `fillEmptyRows`:

```
@grid.GetHtml(
    tableStyle: "grid",
    headerStyle: "head",
    alternatingRowStyle: "alt",
    caption: "Título do gráfico",
    displayHeader: true,
    emptyRowCellValue: "[nulo]",
    fillEmptyRows: true,
    columns: grid.Columns(
        grid.Column("Nome", header: "Nome", format: @<i>@item.Nome</i>),
        grid.Column("Continente", format: @<i>@item.Continente</i>),
        grid.Column("Idioma", format: @<text>** @item.Idioma **</text>)
    )
)
```

Atributos HTML específicos podem ser aplicados com o parâmetro `htmlAttributes`:

```
@grid.GetHtml(
    tableStyle: "grid",
    headerStyle: "head",
    alternatingRowStyle: "alt",
    htmlAttributes: new { @class = "SeletorCSS" },
    columns: grid.Columns(
        grid.Column("Nome", header: "Nome", format: @<i>@item.Nome</i>),
        grid.Column("Continente", format: @<i>@item.Continente</i>),
        grid.Column("Idioma", format: @<text>** @item.Idioma **</text>)
    )
)
```


5.17.4 Paginação de registros

Apaginação dos registros é ativada por padrão. Mas inúmeros parâmetros podem ser definidos, como o número de linhas por página e o tipo de links de paginação, por exemplo, os links serão numéricos (1 2 3 . . . 5) ou texto (Anterior Próximo).

Exemplo:

```
WebGrid grid = new WebGrid(source: Model, rowsPerPage: 3);
@grid.GetHtml(
    tableStyle: "grid",
    headerStyle: "head",
    alternatingRowStyle: "alt",
    mode: WebGridPagerModes.NextPrevious,
    previousText: "Página anterior",
    nextText: "Próxima página",
    columns: grid.Columns(
        grid.Column("Nome", header: "Nome", format: @<i>@item.Nome</i>),
        grid.Column("Continente", format: @<i>@item.Continente</i>),
        grid.Column("Idioma", format: @<text>** @item.Idioma **</text>)
    )
)
```

Quando usamos links de paginação numéricos, temos como determinar o número de links por página com o atributo `numericLinksCount`:

```
WebGrid grid = new WebGrid(source: Model, rowsPerPage: 10);
@grid.GetHtml(
    tableStyle: "grid",
    headerStyle: "head",
    alternatingRowStyle: "alt",
    numericLinksCount: 10,
    columns: grid.Columns(
        grid.Column("Nome", header: "Nome", format: @<i>@item.Nome</i>),
        grid.Column("Continente", format: @<i>@item.Continente</i>),
        grid.Column("Idioma", format: @<text>** @item.Idioma **</text>)
    )
)
```

O resultado vemos na figura 5.9:

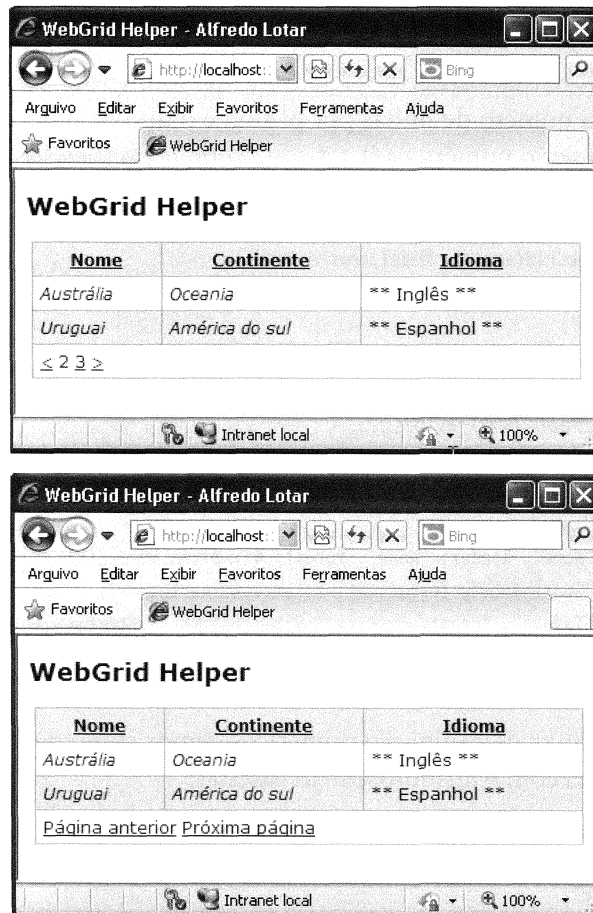


Figura 5.9 – Links de paginação numéricos e de texto.

5.17.5 Seleção de linhas

Para selecionar uma linha, use o método `GetSelectLink`:

```
grid.Column("Select", header: "Selecione", format: <i>@item.GetSelectLink("Select")</i>)
```

Exemplo:

```
WebGrid grid = new WebGrid(source: Model);
@grid.GetHtml(
    columns: grid.Columns(
        grid.Column("Select", header: "Selecione", format:@<i>@item.GetSelectLink("Select")</i>),
        grid.Column("Nome", header: "Nome", format: @<i>@item.Nome</i>),
        grid.Column("Continente", header: "Continente", format: @<i>@item.Continente</i>),
        grid.Column("Idioma", header: "Idioma", format: @<text>** @item.Idioma **</text>)
    )
)
```

Em seguida, capture a informação passada via query string e exiba a página apropriada:

```
@if (grid.HasSelection) {
    @RenderPage("~/Views/Home/GridDetails.cshtml", new { Países = grid.SelectedRow })
}
```

grid.SelectedRow retorna:

AppCapitulo5.Models.Paises

E seus membros podem ser acessados na página-alvo GridDetails.cshtml, conforme o código a seguir:

```
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
    <head>
        <title>GridDetails</title>
    </head>
    <body>
        <div>
            <p>Item selecionado:</p>
            <ul>
                <li><b>Nome:</b> @Page.Paises.Nome</li>
                <br />
                <li><b>Continente:</b> @Page.Paises.Continente</li>
                <br />
                <li><b>Idioma:</b> @Page.Paises.Idioma</li>
            </ul>
        </div>
    </body>
</html>
```

Observe a figura 5.10.

Repare que a página GridDetails.cshtml é exibida na mesma página do WebGrid. Uma alternativa interessante é apresentá-la em uma página modal, ou seja, “flutuando” sobre a página principal.

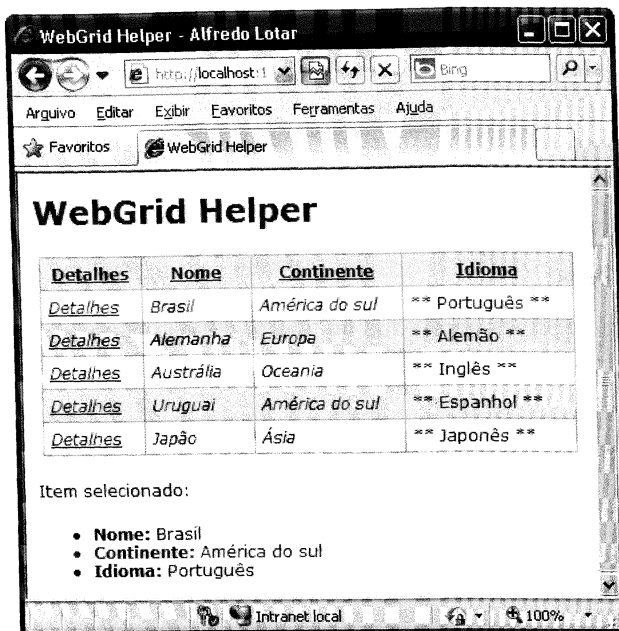


Figura 5.10 – Detalhes de um item.

5.17.6 Extraindo informações do WebGrid

Podemos retornar o índice da linha selecionada:

```
WebGrid grid = new WebGrid(source: Model);
// Restante do código
@if (grid.HasSelection) {
    @grid.SelectedIndex
}
```

O total de páginas:

```
WebGrid grid = new WebGrid(source: Model);
// Restante do código
@if (grid.HasSelection) {
    @grid.PageCount
}
```

E definir ou retornar o índice de determinada página:

```
WebGrid grid = new WebGrid(source: Model);
//Restante do código;
@if (grid.HasSelection) {
    @grid.PageIndex
}
```

5.17.7 WebGrid Helper e Ajax

Para ativar o modo de programação Ajax, defina o parâmetro `ajaxUpdateContainerId`:

```
@{
    WebGrid grid = new WebGrid(source: Model, selectionFieldName: "SelectedRow",
        ajaxUpdateContainerId: "grid");
}
```

E declare o id, grid nesse caso, no local onde os registros do WebGrid helper serão carregados:

```
<div id="grid">
    @{
        @grid.GetHtml()
    }
</div>
```

5.18 WebImage Helper

O WebImage helper exibe no navegador uma imagem previamente definida. Exemplo:

```
WebImage image = new WebImage("~/Content/figura.jpg");
```

O método `Write` exibe a imagem na tela:

```
@image.Write()
```

O construtor `WebImage` aceita nomes de arquivos, um array de bytes ou um tipo stream.

Exemplo:

```
@{
    Layout = null;
}
@{
    System.Drawing.Bitmap objBitmap = new System.Drawing.Bitmap(100, 40);
    System.Drawing.Graphics objGraphics = System.Drawing.Graphics.FromImage(objBitmap);
    System.Drawing.SolidBrush objBrush = new System.Drawing.SolidBrush(System.Drawing.Color.
        White);
    System.Drawing.Font fonte = new System.Drawing.Font("Arial", 14, System.Drawing.FontStyle.
        Bold);
    WebImage image = null;
    MemoryStream ms = null;
    try {
        objGraphics.DrawString("Texto", fonte, objBrush, 5, 8);
        ms = new MemoryStream();
        objBitmap.Save(ms, System.Drawing.Imaging.ImageFormat.Gif);
        image = new WebImage(ms);
    }
}
```

```

        finally {
            if (objBitmap != null) { objBitmap.Dispose();}
            if (objBitmap != null) { ms.Dispose(); }
        }
    }
}
<!DOCTYPE html>
<html>
    <head>
        <title>WebImage Helper</title>
    </head>
    <body>
        <div>
            <p>
                @image.Write()
            </p>
        </div>
    </body>
</html>

```

Ou seja, você cria suas próprias imagens via código C# ou as extrai de um banco de dados e exibe no navegador.

É possível também capturar uma figura carregada via upload:

```
WebImage.GetImageFromRequest(postedFileName: "Nomedoarquivo");
```

Se necessário, aplique uma marca d'água na imagem por intermédio do método `AddImageWatermark`:

```

WebImage image = new WebImage("~/Content/figura.jpg");
image.AddImageWatermark("~/Content/figura1.jpg", width: 100, height: 100,
    horizontalAlign: "center", verticalAlign: "middle");
<p>
    @image.Write()
</p>

```

E `AddTextWatermark`:

```

WebImage image = new WebImage("~/Content/figura.jpg");
image.AddTextWatermark("Autor: Alfredo Lotar", fontColor: "blue", horizontalAlign: "center",
    verticalAlign: "middle");
<p>
    @image.Write()
</p>

```

Recurso interessante quando publicamos fotos na Internet e queremos manter em evidência a origem da imagem.

5.19 Chart Helper

O Chart helper exibe dados em um gráfico. É possível exibir as informações em mais de 30 tipos de gráficos, incluindo os tipos usados pelo Microsoft Excel.

Na figura 5.11 exibimos os principais tipos de gráficos usados.

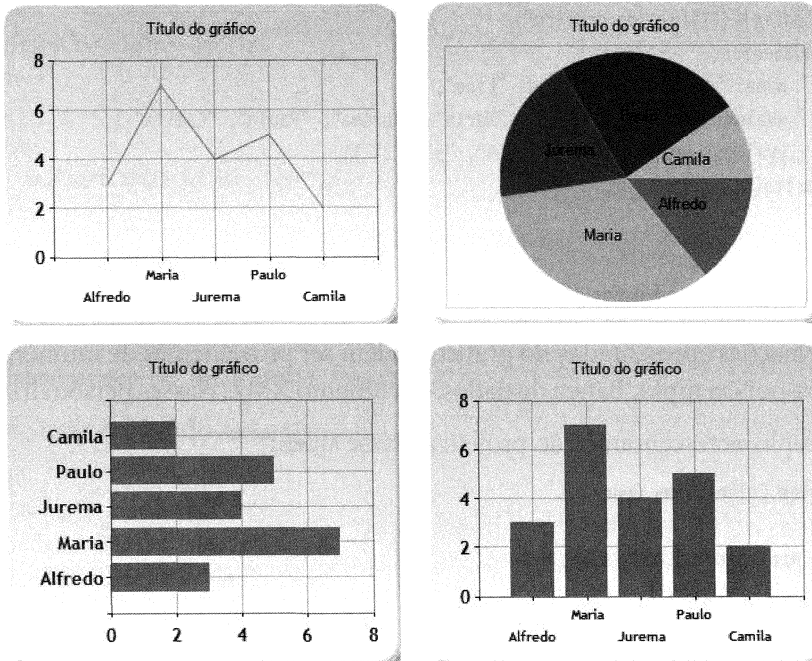


Figura 5.11 – Tipos de gráficos do Chart helper.

Criar um novo gráfico é muito fácil. Basta criar uma instância da classe `Chart` e definir as dimensões do gráfico.

```
var myChart = new Chart(width: 300, height: 250)
```

O método `AddTitle` determina o título do gráfico:

```
.AddTitle("Título do gráfico")
```

E o método `AddSeries` acrescenta os dados ao gráfico. Os parâmetros `xValue` e `yValues` representam, respectivamente, os eixos `x` e `y`. O parâmetro `chartType` define o tipo de gráfico usado e o parâmetro `name` nomeia a série com um id exclusivo. Exemplo:

```
.AddSeries(name: "Alunos", chartType: "Line", xValue: new[] { "Alfredo", "Maria", "Jurema",  
    "Paulo", "Camila" }, yValues: new[] { "3", "7", "4", "5", "2" })
```

O método `AddLegend` adiciona uma legenda ao gráfico.

Para exibir o gráfico no navegador, use o método `Write`:

```
.Write();
```

Ou:

```
.Write(format: "gif");
```

Observe o código a seguir:

```
@{
    var myChart = new Chart(width: 300, height: 250)
        .AddTitle("Título do gráfico")
        .AddSeries(
            name: "Alunos", chartType: "Line",
            xValue: new[] { "Alfredo", "Maria", "Jurema", "Paulo", "Camila" },
            yValues: new[] { "3", "7", "4", "5", "2" })
        .Write();
}
```

5.19.1 Usando outras fontes de dados

As informações apresentadas no gráfico podem ser provenientes de inúmeras fontes de dados, por exemplo: Banco de dados, documento XML, classes personalizadas etc.

Por exemplo, acrescentamos ao projeto a classe Alunos:

```
using System.Collections.Generic;
```

```
namespace AppCapitulo5.Models {
    public class Alunos {
        public string Nome { get; set; }
        public int Nota { get; set; }
        public static List<Alunos> GetAlunos() {
            return new List<Alunos> {
                new Alunos {Nome="Alfredo", Nota=3},
                new Alunos {Nome="Maria", Nota=7},
                new Alunos {Nome="Jurema", Nota=4},
                new Alunos {Nome="Paulo", Nota=5},
                new Alunos {Nome="Camila", Nota=2}
            };
        }
    }
}
```

E, em seguida, carregamos os dados da classe no Chart helper:

```
@using AppCapitulo5.Models
@{
    Layout = null;
}
@{
    var data = Alunos.GetAlunos();
```



```

var myChart = new Chart(width: 300, height: 250)
    .AddTitle("Título do gráfico")
    .DataBindTable(dataSource: data, xField: "Nome");
}
<!DOCTYPE html>
<html>
    <head>
        <title>Chart helper</title>
    </head>
    <body>
        <div>
            @myChart.Write(format: "gif");
        </div>
    </body>
</html>

```

Em vez do método `DataBindTable`:

```
.DataBindTable(dataSource: data, xField: "Nome");
```

Você pode usar o método `AddSeries`:

```

.AddSeries(
    name: "Default", chartType: "Line",
    xValue: data, xField: "Nome",
    yValues: data, yFields: "Nota"
);

```

5.19.2 Exibindo gráficos em uma página

A tag HTML `img` carrega o gráfico dentro de outra página.

Exemplo:

```
<p></p>
```

Repare que carregamos o view `Chart1` que contém o gráfico e não uma imagem.

5.19.3 Personalizar a aparência

Para personalizar a aparência do gráfico, defina o parâmetro `theme` como: `Vanilla`, `Blue`, `Green`, `Yellow` ou `Vanilla3D`.

Exemplo:

```
var myChart = new Chart(width: 300, height: 250, theme: ChartTheme.Green)
```

Observe a figura 5.12:

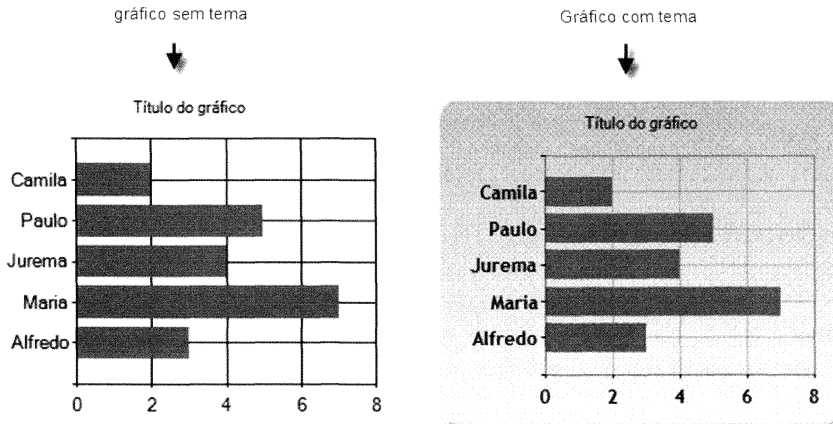


Figura 5.12 – Diferença entre um gráfico com tema e outro sem.

5.19.4 Salvando um gráfico

O método Save salva o gráfico no HD:

```
@{
    var data = Alunos.GetAlunos();
    var myChart = new Chart(width: 300, height: 250)
        .AddTitle("Titulo do gráfico")
        .AddSeries(
            name: "Default", chartType: "Bar",
            xValue: data, xField: "Nome",
            yValues: data, yFields: "Nota"
        )
        .Save(@"c:\teste\grafico.jpeg");
}
```

Claro que é necessário permissão de gravação no diretório.

O gráfico também pode ser salvo como um documento XML e posteriormente recuperado:

```
@using AppCapitulo5.Models
@{
    Layout = null;
}
@{
    Chart xmlChart=null;
    var arquivo = "/abc.xml";
    if (File.Exists(Server.MapPath(arquivo))) {
        xmlChart = new Chart(width: 600, height: 400, themePath: arquivo);
    }
    else
    {
```

```

var data = Alunos.GetAlunos();
xmlChart = new Chart(width: 300, height: 250, theme: ChartTheme.Green)
    .AddTitle("Título do gráfico")
    .AddSeries(
        name: "Default", chartType: "Bar",
        xValue: data, xField: "Nome",
        yValues: data, yFields: "Nota"
    )
    .SaveXml(path: Server.MapPath(arquivo));
}
}
<!DOCTYPE html>
<html>
    <head>
        <title>Gráfico com o método SaveXml</title>
    </head>
    <body>
        <div>
            @xmlChart.Write()
        </div>
    </body>
</html>

```

5.19.5 Colocando um gráfico no cache

O método `SaveToCache` salva o gráfico no cache:

```
cacheChart.SaveToCache(key: chave, minutesToCache: 2, slidingExpiration: true);
```

O parâmetro `minutesToCache` define o tempo em que o cache está ativo. E o parâmetro `slidingExpiration` determina se os dados permanecem ou não no cache enquanto houver requisições.

5.19.5.1 Obtendo dados do cache

O método `GetFromCache` “pega” o gráfico do cache:

```
var cacheChart = Chart.GetFromCache(key: chave);
```

Enquanto `WriteFromCache` exibe o gráfico no navegador:

```
Chart.WriteFromCache(chave);
```

Observe o código a seguir:

```

@{
    var chave = Request["key"];
    if (chave != null) {
        var cacheChart = Chart.GetFromCache(key: chave);
        if (cacheChart == null) {
            cacheChart = new Chart(300, 250, theme: ChartTheme.Green);

```

```

cacheChart.AddTitle("Título do gráfico");
cacheChart.AddSeries(
    name: "Alunos", chartType: "Line",
    xValue: new[] { "Alfredo", "Maria", "Jurema", "Paulo", "Camila" },
    yValues: new[] { "3", "7", "4", "5", "2" });

cacheChart.SaveToCache(key: chave,
    minutesToCache: 2,
    slidingExpiration: true);
cacheChart.Write();
}
Chart.WriteFromCache(chave);
}
}

```

Insira o código anterior no arquivo *Chart3.cshtml* e carregue o gráfico em um view qualquer da seguinte forma:

```

@{
    ViewBag.Title = "Título da página";
}
<div>
    <p></p>
</div>

```

5.20 WebMail Helper

O *WebMail* helper envia um e-mail a partir de uma página de um website. Um website ASP.NET envia e-mails para os usuários em diversas situações, por exemplo, para:

- Enviar o conteúdo de um formulário de contato.
- Enviar lembretes para os usuários.
- Enviar newsletters.
- Enviar confirmação de cadastros.
- Enviar lembretes de senha ou novas senhas.
- Enviar o conteúdo de uma página.

A propriedade *SmtpServer* do *WebMail* helper define o nome do servidor SMTP usado para transmitir o e-mail:

```
WebMail.SmtpServer = "localhost";
```

Ou:

```
WebMail.SmtpServer = "smtp.website.com.br";
```

A propriedade `SmtpPort` define o número da porta e a propriedade `EnableSsl` ativa ou desativa a transmissão segura via SSL:

```
WebMail.SmtpPort = 25;
WebMail.EnableSsl = true;
```

As credenciais usadas para transmitir o e-mail podem ser definidas explicitamente:

```
WebMail.UserName = "Idusuário";
WebMail.Password = "Senha";
```

Ou use as credenciais-padrão do Windows:

```
WebMail.SmtpUseDefaultCredentials = true;
```

O método `Send` envia o e-mail. É aqui que definimos o e-mail de destino, o assunto, o corpo da mensagem, os arquivos anexos e também os cabeçalhos da mensagem:

```
var arquivo = new string[] { file1 };
WebMail.Send(to: email,
    subject: assunto,
    body: mensagem,
    isBodyHtml:true,
    filesToAttach: arquivo
);
```

5.20.1 Enviando um e-mail com anexos

Para enviar um e-mail no ASP.NET MVC, primeiramente definimos o código nos métodos de controlador. Usamos no exemplo dois métodos `Enviar`:

```
public ActionResult Enviar() {
    Response.Cache.SetCacheability(HttpCacheability.NoCache);
    Response.Cache.SetAllowResponseInBrowserHistory(false);
    return View();
}
```

[HttpPost]

```
public ActionResult Enviar(string email, string assunto, string mensagem, string file1) {
    try {
        WebMail.SmtpServer = "localhost";
        WebMail.SmtpPort = 25;
        // WebMail.EnableSsl = true;
        WebMail.From = "emailorigem@website.com";

        // WebMail.UserName = "Idusuário";
        // WebMail.Password = "Senha";
        WebMail.SmtpUseDefaultCredentials = true;
```

```

        var arquivo = new string[] { file1 };

        WebMail.Send(to: email,
            subject: assunto,
            body: mensagem,
            isBodyHtml:true,
            filesToAttach: arquivo
        );
        return RedirectToAction("EmailSend");
    }
    catch (Exception ex) {
        ModelState.AddModelError("Error_Form", ex.Message);
    }
    return View();
}

```

O trecho de código a seguir, inserido no primeiro método `Enviar`, evita que o formulário seja postado várias vezes com o mesmo conteúdo:

```

Response.Cache.SetCacheability(HttpCacheability.NoCache);
Response.Cache.SetAllowResponseInBrowserHistory(false);

```

Quando o e-mail é enviado com sucesso, redirecionamos o usuário para o view `EmailSend`.

```

return RedirectToAction("EmailSend");

```

Se o e-mail não for enviado por um motivo qualquer, o método `AddModelError` exibe uma mensagem de erro no topo do formulário.

```

catch (Exception ex) {
    ModelState.AddModelError("Error_Form", ex.Message);
}

```

Obs.: o método `AddModelError` é abordado no capítulo 8.

Após a definição do código nos métodos de controlador, criamos o formulário de envio.

Exemplo:

```

@using AlfredoLotar.Livro.AspnetMvc
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
    <head>
        <title>WebMail Helper</title>
    </head>

```

```

<body>
    <div>
        <p style="color: red">@Html.ValidationMessage("Error_Form")</p>
        @using (Html.BeginForm("Enviar", "Home")) {
            <table>
                <tr>
                    <td style="text-align: right;" colspan="2">
                        E-mail:
                    </td>
                    <td>
                        @Html.TextBox("email", "", new { @maxlength = "50",
                            @style = "width: 367px;" })
                    </td>
                </tr>
                <tr>
                    <td style="text-align: right;" colspan="2">
                        Assunto:
                    </td>
                    <td>
                        @Html.TextBox("assunto", "", new { @maxlength = "100",
                            @style = "width: 367px;" })
                    </td>
                </tr>
                <tr>
                    <td style="text-align: right;" colspan="2">
                        Mensagem:
                    </td>
                    <td>
                        @Html.TextArea("mensagem", "", new { @cols = "20", @rows = "5",
                            @style = "width: 367px; height: 178px" })
                    </td>
                </tr>
                <tr>
                    <td style="text-align: right;" colspan="2">
                        Arquivo:
                    </td>
                    <td>
                        @Html.File(@"C:\Livro\AppCapitulo5\file.txt")
                    </td>
                </tr>
                <tr>
                    <td style="text-align: right;" colspan="2">
                    </td>
                    <td>
                        <input id="btnEnviar" type="submit" value="Enviar e-mail" />
                    </td>
                </tr>
            </table>
        }
    </div>
</body>

```

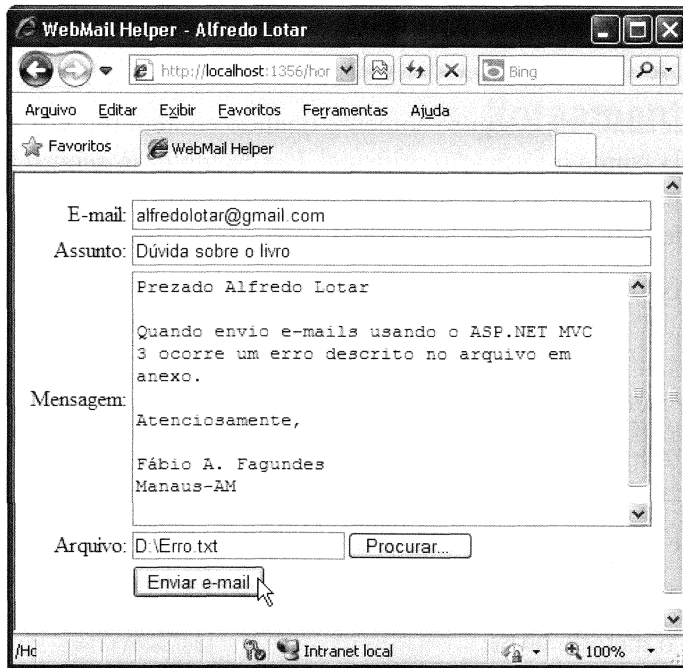



Figura 5.13 – Formulário de envio de e-mail com anexos.

5.20.2 Acrescentando cabeçalhos à mensagem

Algumas mensagens exigem que o usuário confirme o recebimento do e-mail. Você pode acrescentar ao método `Send` do `WebMail` helper o parâmetro `additionalHeaders` e definir o cabeçalho de confirmação da mensagem.

Exemplo:

```
WebMail.Send(to: email,
    subject: assunto,
    body: mensagem,
    filesToAttach: arquivo,
    isBodyHtml: true, additionalHeaders: new string[]{"Disposition-Notification-To:emailConfirmacao@website.com"}
);
```

5.21 Crypto Helper

`Crypto` helper é relativamente simples, mas útil quando necessitamos armazenar e verificar um hash de senhas ou gerar um hash qualquer.

Hash representa um valor único e corresponde a um conjunto de bytes. Em outras palavras, você transforma uma informação legível em algo indecifrável.

Por exemplo, meu nome: Alfredo Lotar se transforma no hash:

F02E8A58A04AB6B7628CFA94E3CDD1E8234BAA2F24A35EE407260554A84890F9

Lembrando que um hash é seguro somente quando contém um salt – informação aleatória cifrada com a informação principal. Por exemplo, ao gerar o hash do meu nome, incluímos uma senha, ou um valor qualquer. Assim, para um usuário mal-intencionado descobrir o conteúdo original, é necessário descobrir também o salt.

Gerar um hash é uma tarefa simples para o Crypto helper.

Exemplo:

```
@{
    string salt = Crypto.GenerateSalt();
    @Crypto.Hash("Alfredo Lotar" + salt)
}
```

Gera a seguinte saída:

4077383C8FA49A74A432810D6B505CE4E0AFA2A12BA11C6FACC92F5921802D8D

Se preferir, use o método SHA256:

```
@{
    string salt = Crypto.GenerateSalt();
    @Crypto.SHA256("Alfredo Lotar" + salt)
}
```

O Crypto helper possui um mecanismo de geração de hash de senhas.

Exemplo:

```
@{
    @Crypto.HashPassword("Senha")
}
```

O valor gerado:

AGZr+IoupN0n0kUPYg3RWjUMQjPFcA3NSXxZvQ409CMEnAz41eYumLJE14LT676mzQ==

Pode ser armazenado em uma tabela de um banco de dados. Quando o usuário faz o login, basta comparar o valor digitado com o hash armazenado:

```
@{
    @Crypto.VerifyHashedPassword("AGZr+IoupN0n0kUPYg3RWjUMQjPFcA3NSXxZvQ409CMEnAz41eYumLJE14LT676mzQ==", "Senha")
}
```

Obs.: a partir deste capítulo, usaremos somente views com Razor.

Roteamento de URLs

O roteamento de URLs consiste na definição e requisição de termos significativos e parâmetros declarados no arquivo `global.asax`.

O roteamento de URLs visa a facilitar a vida do usuário, por exemplo, em vez de digitar URLs grandes e complicadas, o usuário digita algo que faz sentido para ele, como:

```
http://www.novatec.com.br/vendas  
http://www.novatec.com.br/livros  
http://www.novatec.com.br/download
```

No ASP.NET MVC, o sufixo `vendas`, `livros` e `download` pode ser um método de controlador ou o nome de um view. Nas primeiras versões do ASP.NET, o usuário acessava arquivos `.aspx`.

Exemplo:

```
http://www.novatec.com.br/vendas.aspx  
http://www.novatec.com.br/livros.aspx  
http://www.novatec.com.br/download.aspx
```

6.1 Modelo de URLs

Um modelo de URLs contém valores literais e variáveis cercados por chaves (`{}`) e separados por uma barra (`/`).

Exemplo:

```
pedidos/{dia}/{mes}/{ano}  
{idioma}-{pais}/{action}  
produtos/view/{categoria}  
{controller}/{action}/{id}
```

O caractere asterisco (*) antes da variável:

```
Categorias/{*NomeCategoria}
```

Permite que a URL suporte seguimentos extras daqueles definidos no modelo. A URL a seguir é normal, sem seguimentos extras:

`http://localhost/website/categorias/bebidas`

No entanto, a URL que exhibe a categoria carnes/aves tem um seguimento extra, aves:

`http://localhost/website/categorias/carnes/aves`

O seguimento extra é considerado parte do último seguimento.

Quando criamos uma nova aplicação ASP.NET MVC, automaticamente é adicionado ao arquivo `global.asax` o seguinte modelo de URL:

```
public static void RegisterRoutes(RouteCollection routes) {
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

    routes.MapRoute(
        "Default", // Route name
        "{controller}/{action}/{id}", // URL with parameters
        new { controller = "Home", action = "Index", id = UrlParameter.Optional } // Parameter defaults
    );
}
```

Obs.: uma nova aplicação ASP.NET MVC supõe que existe um controlador `HomeController`. Se você não pretende usar um com esse nome, altere o nome no arquivo `global.asax`.

O método `MapRoute` registra os modelos de URLs usados pela aplicação ASP.NET MVC.

A seguir, descrevemos os principais parâmetros do método `MapRoute`:

Parâmetro	Descrição
Name	Nome da rota.
URL	Define o modelo de URLs usado pela rota.
Defaults	Define os valores-padrão ou opcionais usados pela rota.
Constraints	Determina as restrições.

Obs.: aplicações ASP.NET Web Forms utilizam o método `MapPageRoute` em vez de `MapRoute`.

No parâmetro `Defaults` definimos os valores opcionais:

```
new { controller = "Home", action = "Index", id = UrlParameter.Optional }
```

ou valores-padrão:

```
new { controller = "Home", action = "Index", id = 5 }
```

Além do modelo de URL predefinido adicionado ao projeto, você pode acrescentar modelos customizados, por exemplo:

```
routes.MapRoute(
    "produtosview ",
    "produtos/view/{categoria}",
    new { controller = "produtos", action = "view" }
);
```

Que combina com a URL a seguir:

<http://localhost/website/produtos/view/bebidas>

Após a definição dos modelos de URLs, registramos as rotas no evento `Application_Start` do arquivo `global.asax`:

```
protected void Application_Start() {
    RegisterRoutes(RouteTable.Routes);
}
```

6.2 Restrições

Quando definimos os modelos de URLs, podemos determinar restrições quanto ao formato de URLs permitidas na aplicação.

Com o auxílio de expressões regulares, facilmente restringimos o formato do id da página de detalhes da aplicação. Exemplo:

```
routes.MapRoute(
    "Default",
    "{controller}/{action}/{id}",
    new { controller = "Home", action = "Index", id = 0 }, new { id = @"\d{1,2}" }
);
```

No exemplo, o identificador pode conter, no mínimo, um e, no máximo, dois dígitos:

<http://localhost/AppCapitulo6/Home/Details/5>

<http://localhost/AppCapitulo6/Home/Details/15>

6.2.1 Método `HttpMethodConstraint`

O método `HttpMethodConstraint` define restrições quanto à operação permitida por um modelo de URL específico.

```
routes.MapRoute(
    "Default",
    "{controller}/{action}/{id}",
    new { controller = "Home", action = "Create", id = UrlParameter.Optional },
    new { method = new HttpMethodConstraint("POST") }
);
```

O método `Create` é executado somente por uma operação `HttpPost`.

6.2.2 Restrições personalizadas

Se necessário, podemos criar métodos personalizados para restringir o uso de determinada rota.

A classe que contém o método deve ser derivada de `IRouteConstraint`.

Exemplo:

```
using System.Web;
using System.Web.Routing;

namespace AppCapitulo6.Models {
    public class SecureConnectionConstraint : IRouteConstraint {
        public bool Match (HttpContextBase httpContext, Route route, string parameterName,
            RouteValueDictionary values, RouteDirection routeDirection) {
            return httpContext.Request.IsSecureConnection;
        }
    }
}
```

A linha:

```
return httpContext.Request.IsSecureConnection;
```

Restringe o acesso à rota a conexões seguras. Por exemplo, se você deseja restringir o acesso à rota para usuários autenticados, substitua:

```
return httpContext.Request.IsSecureConnection;
```

Por:

```
return httpContext.Request.IsAuthenticated;
```

Após criar a classe `SecureConnectionConstraint`, aplique-a desta forma à rota:

```
routes.MapRoute(
    "Default",
    "{controller}/{action}/{id}",
    new { controller = "Home", action = "Create", id = UrlParameter.Optional },
    new { method = new SecureConnectionConstraint() }
);
```

6.3 Definindo URLs

Os links das páginas do ASP.NET MVC podem ser gerados da forma tradicional com a tag HTML `<a>`:

```
<a href="home/details">detalhes</a>
<a href="http://www.novatec.com.br">novatec</a>
```

```
<a href="http://www.novatec.com.br">  
    
</a>
```

Ou com métodos específicos como o `ActionLink`.

6.3.1 Gerando links com `ActionLink`

O método `ActionLink` gera o link usando o controlador, o método de controlador e parâmetros opcionais.

Exemplo:

```
<html>  
  <body>  
    <div>  
      @Html.ActionLink("Texto do link", "Details", "Home")  
    </div>  
  </body>  
</html>
```

O mecanismo de exibição ASPX usa `<%: %>`:

```
<html>  
  <body>  
    <div>  
      <%: Html.ActionLink("Texto do link", "Details", "Home") %>  
    </div>  
  </body>  
</html>
```

Em ambos os casos, o navegador interpreta a linha como:

```
<a href="/Home/Details">Texto do link</a>
```

Se o link é acessado dentro da mesma hierarquia de arquivos, podemos simplesmente escrever:

```
@Html.ActionLink("Texto do link", "Details")
```

Exemplo:

```
<html>  
  <body>  
    <div>  
      @Html.ActionLink("Texto do link", "Details")  
    </div>  
  </body>  
</html>
```

6.3.1.1 Passando parâmetros

Você pode passar informações para a URL que posteriormente podem ser acessadas via query string:

```
@Html.ActionLink("Texto do link", "Details", "Home", new { id = 5, page = 2, cor = "azul" }, null)
@Html.ActionLink("Texto do link", "Details", "Home", new { id = 5, page = 2, cor = "azul" },
    new { target = "_blank" })
```

Exemplo:

```
<html>
  <body>
    <div>
      @Html.ActionLink("Texto do link", "Details", "Home", new { id = 5, page = 2,
        cor = "azul" }, null)
    </div>
  </body>
</html>
```

O navegador interpreta como:

```
<a href="/Home/Details/5?page=2&cor=azul">Texto do link</a>
```

6.3.1.2 ActionLink e CSS:

Referências a seletores CSS podem ser definidas facilmente por meio da propriedade class:

```
@Html.ActionLink("Texto do link", "Details", "Home", new { id = 5, page = 2, cor = "azul" },
    new { @class = "seletorCSS", target = "_blank" })
```

Ou:

```
<%:Html.ActionLink("Texto do link", "Details", "Home", new { id = 5, page = 2, cor = "azul" },
    new { @class = "seletorCSS", target = "_blank" })%>
```

Se preferir, omita a propriedade target:

```
@Html.ActionLink("Texto do link", "Details", "Home", new { id = 5, page = 2, cor = "azul" },
    new { @class = "seletorCSS" })
<%:Html.ActionLink("Texto do link", "Details", "Home", new { id = 5, page = 2, cor = "azul" },
    new { @class = "seletorCSS" })%>
```

6.3.2 Gerando links com RouteLink

Linka uma URL com base em uma rota predefinida:

```
@Html.RouteLink("Clique aqui", new { controller = "Home", action = "Details" })
```


Exemplo:

```
<html>
  <body>
    <div>
      <p>@Html.RouteLink("Clique aqui", new { controller = "Home", action = "Details" })
    </p>
    </div>
  </body>
</html>
```

Retorna ao navegador:

```
<a href="/Home/Details">Clique aqui</a>
```

O modo anterior é semelhante ao usado pelo método `ActionLink`. Se preferir, crie um link para uma rota específica:

```
<p>@Html.RouteLink("Clique aqui.", "Detalhes") </p>
```

Definida no arquivo `global.asax`:

```
routes.MapRoute(
    "Detalhes",
    "Home/Details/{id}",
    new { action = "Details", id = UrlParameter.Optional }
);
```

O navegador novamente interpreta a linha como:

```
<a href="/Home/Details">Clique aqui.</a>
```

Usar a rota em vez do nome e do método de controlador ao definir o link é interessante, pois facilita a manutenção da aplicação e permite maior flexibilidade ao designer.

6.3.3 Redirecionamentos

No ASP.NET MVC há várias formas de redirecionamento. Podemos redirecionar o usuário para um método de controlador:

```
return RedirectToAction("Index");
return RedirectToAction("Detalhes", new { id = 5 });
```

Exemplo:

```
public ActionResult Details(int? id) {
    if (!id.HasValue)
        return RedirectToAction("Index");
    ViewBag.Mensagem = "Detalhes id = " + id;
    return View();
}
```

Ou use o método `RedirectToActionPermanent`:

```
public ActionResult Details(int? id) {  
    if (!id.HasValue)  
        return RedirectToActionPermanent("Index");  
    ViewBag.Mensagem = "Detalhes id = " + id;  
    return View();  
}
```

Para uma rota específica, use:

```
public ActionResult Details(int? id) {  
    if (!id.HasValue)  
        return RedirectToRoute("Detalhes");  
    // return RedirectToRoute("Detalhes", new { id = 5 });  
    ViewBag.Mensagem = "Detalhes id = " + id;  
    return View();  
}
```

Ou, se preferir, use o método `RedirectToRoutePermanent`:

```
public ActionResult Details(int? id) {  
    if (!id.HasValue)  
        return RedirectToRoutePermanent("Detalhes");  
    ViewBag.Mensagem = "Detalhes id = " + id;  
    return View();  
}
```

Para redirecionar o usuário a um website específico, use o método `Redirect`:

```
public ActionResult Visitar() {  
    return Redirect("http://www.novatec.com.br");  
}
```

Ou o método `RedirectPermanent`:

```
public ActionResult Visitar() {  
    return RedirectPermanent("http://www.novatec.com.br");  
}
```

6.3.4 Obtendo uma URL

Para elaborar uma URL a partir de uma rota ou método de controlador por intermédio dos membros da classe `Url`, use o método `Action`:

```
@Url.Action("List")  
@Url.Action("Details", new { id = 10 })
```

Exemplo:

```
<a href=@Url.Action("Details", new { id = 10 })>Clique aqui</a>
```

Ou use o método `RouteUrl`:

```
@Url.RouteUrl("NomeDaRota", new { id = 35 })
```

Exemplo:

```
<a href=@Url.RouteUrl("DefaultCreate", new { id = 35 })>Clique aqui</a>
```

6.4 Áreas

Uma aplicação ASP.NET MVC pode ser dividida em áreas, que fornecem um modo de dividir uma aplicação web em pequenas seções funcionais. Por exemplo, uma área contém os controladores, view, model e os arquivos referentes à administração do website, outra área contém a estrutura referente aos produtos exibidos.

A ideia central é a organização e o isolamento das diferentes seções da aplicação.

Lembrando que cada área contém sua própria estrutura de arquivos e diretórios.

Para criar uma nova área para um website, na janela **Solution Explorer**, clique com o botão direito do mouse no nome do projeto e acesse **Add** e, em seguida, clique em **Area...**. Observe a figura 6.1:

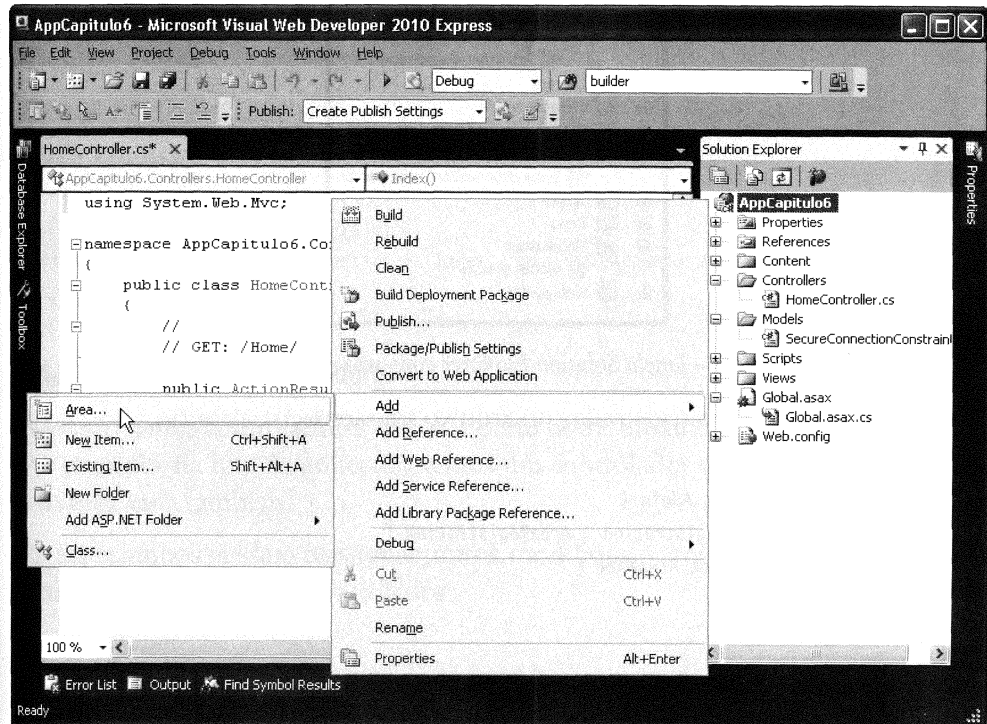


Figura 6.1 – Passos para acrescentar uma nova área.

Na caixa de texto **Area Name**, digite **Admin** e clique em **Add**. Observe a figura 6.2:

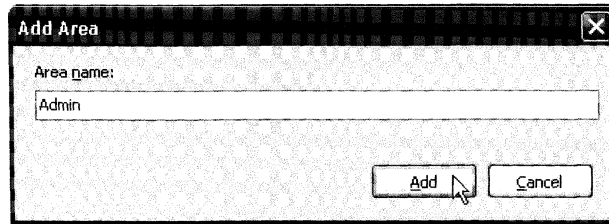


Figura 6.2 – Caixa de diálogo Add Area.

Ao clicar em **Add**, geramos a estrutura de diretórios apresentada na figura 6.3:

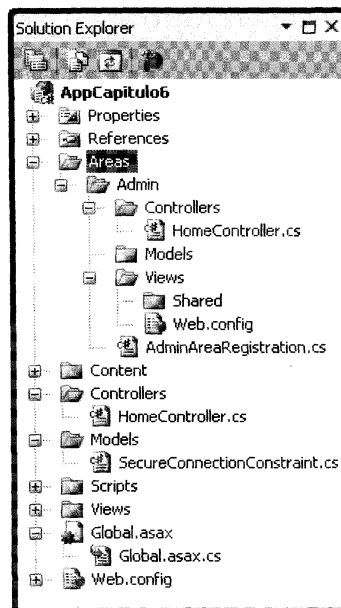


Figura 6.3 – Janela Solution Explorer contendo uma nova Área.

As rotas da área Admin são registradas no arquivo AdminAreaRegistration.cs:

```
using System.Web.Mvc;
namespace AppCapitulo6.Areas.Admin {
    public class AdminAreaRegistration : AreaRegistration {
        public override string AreaName {
            get {
                return "Admin";
            }
        }
        public override void RegisterArea(AreaRegistrationContext context) {
            context.MapRoute(
                "Admin_default",
```

```

        "Admin/{controller}/{action}/{id}",
        new { action = "Index", id = UrlParameter.Optional }
    );
}
}
}

```

Quando acrescentamos uma nova área, não conseguimos acessar os métodos do controlador desta forma:

`http://localhost:2676/admin`

Ou seja, é necessário o caminho completo:

`http://localhost:2676/admin/home`

Para contornar esse problema, altere o arquivo `AdminAreaRegistration.cs`. Acrescente a rota:

```

context.MapRoute(
    "Admin_default",
    "Admin/{controller}/{action}/{id}",
    new { action = "Index", id = UrlParameter.Optional }
);

```

O nome do controlador:

`controller = "Home"`

Exemplo:

```

context.MapRoute(
    "Admin_default",
    "Admin/{controller}/{action}/{id}",
    new { controller = "Home", action = "Index", id = UrlParameter.Optional }
);

```

6.4.1 Conflitos entre controladores

Não raro temos controladores com o mesmo nome em diferentes áreas da aplicação. Por exemplo, na raiz da aplicação, temos um controlador denominado `HomeController` e na área `Admin` também.

Para que ambos possam funcionar juntos, modifique o arquivo `Global.asax`. Acrescente a namespace usada pelo controlador:

```

routes.MapRoute(
    "Default",
    "{controller}/{action}/{id}",
    new { controller = "Home", action = "Index", id = UrlParameter.Optional },
    new[] { "AppCapitulo6.Controllers" }
);

```

6.4.2 Link para diferentes áreas

Para linkar um método de controlador em outra área, especifique explicitamente o nome da área.

Exemplo:

```
@Html.ActionLink("Administração", "Index", "Home", new { area = "Admin" }, null)
```

Se for necessário, passe parâmetros à URL:

```
@Html.ActionLink("Administração", "Index", "Home", new { id = 5, area = "Admin" }, null)
```

A formatação com seletores CSS também pode ser aplicada:

```
@Html.ActionLink("Administração", "Index", "Home", new { id = 5, area = "Admin" },  
    new { @class = "seletorCSS" })
```

6.4.3 Link para a raiz da aplicação

Para acessar um recurso na raiz da aplicação a partir de uma área específica, defina o nome da área com uma string vazia:

```
@Html.ActionLink("Página inicial", "Index", "Home", new { area = string.Empty }, null)
```

HTML Helpers

HTML helpers são métodos que exibem no navegador texto simples ou elementos HTML de um formulário, ou um elemento HTML qualquer.

Um HTML helpers pode ser simples e exibir apenas uma string ou sofisticado, como os usados para exibir vários tipos de gráficos.

7.1 Desenvolvendo uma nova aplicação

Para testar os exemplos deste capítulo, inicie o Visual Studio. Acesse o menu **File** e clique em **New Project ...**.

Na caixa de diálogo **New Project**, selecione **Web > Visual Studio 2012 > ASP.NET MVC 4 Web Application**. Nomeie o projeto como **AppCapítulo7**. Clique em **OK**.

Em seguida, na caixa de combinação **View Engine**, selecione o mecanismo de exibição **Razor**.

Adicione ao projeto o controlador **HomeController**.

No diretório **Models**, crie e adicione entidades para um arquivo **.edmx** a partir do banco de dados **Northwind**. Siga os passos descritos no tópico “4.3 ADO.NET Entity Framework”.

7.2 Formulários

Há três formas diferentes de se criar formulários no ASP.NET MVC. A primeira usa elementos HTML e é processada no cliente, recomendada por motivos de performance.

Exemplo:

```
<!DOCTYPE html>
<html>
  <head runat="server">
    <title>Formulário de cadastro com elementos HTML</title>
  </head>
  <body>
    <div>
```

```

<fieldset style="width: 460px">
  <legend>Cadastro</legend>
  <table style="width: 100%;">
    <tr>
      <td style="text-align: right; width: 69px;">
        Nome:
      </td>
      <td colspan="3">
        <input id="Text1" style="width: 240px;" type="text" />
      </td>
    </tr>
    <tr>
      <td style="text-align: right; width: 69px;">
        End:
      </td>
      <td colspan="3">
        <input id="Text2" style="width: 240px;" type="text" />
      </td>
    </tr>
    <tr>
      <td style="text-align: right; width: 69px;">
        Cidade:
      </td>
      <td style="width: 125px;">
        <input id="Text3" type="text" />
      </td>
      <td style="text-align: right; width: 41px;">
        Estado:
      </td>
      <td>
        <select id="Select1" name="D1">
          <option>RS</option>
          <option>SC</option>
          <option>RJ</option>
          <option>SP</option>
          <option>PR</option>
        </select>
      </td>
    </tr>
    <tr>
      <td style="width: 69px; text-align: right; height: 30px;" valign="top">
      </td>
      <td colspan="3" style="height: 30px;">
        <input id="Submit1" type="submit" value="OK" />
        <input id="Submit2" type="submit" value="Cancelar" />
      </td>
    </tr>
  </table>
</fieldset>

```



```

        </tr>
    </table>
</fieldset>
</div>
</body>
</html>

```

A saída vemos na figura 7.1:

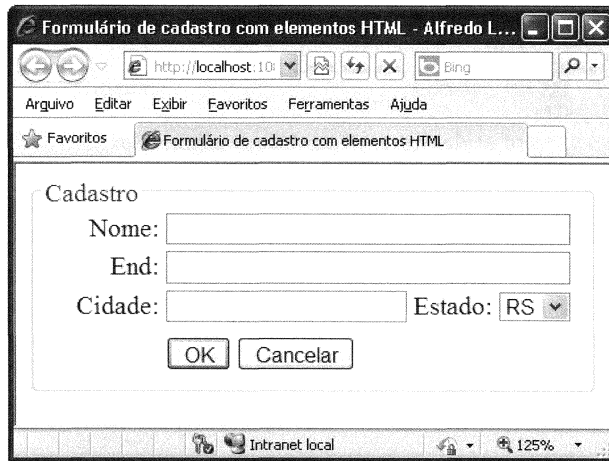


Figura 7.1 – Formulário de cadastro com elementos HTML.

O segundo modo usa Web Server Controls e é processado no servidor Web.

Exemplo:

```

<!DOCTYPE html>
<html>
    <head runat="server">
        <title>Formulário de cadastro com Server Controls </title>
    </head>
    <body>
        <form runat="server">
            <div>
                <fieldset style="width: 330px">
                    <legend>Cadastro</legend>
                    <table style="width: 100%;">
                        <tr>
                            <td style="text-align: right; width: 69px;">
                                Nome:
                            </td>
                            <td colspan="3">
                                <asp:TextBox ID="TextBox1" runat="server" Style="width: 240px;">
                                </asp:TextBox>

```

```

        </td>
    </tr>
    <tr>
        <td style="text-align: right; width: 69px;">
            End:
        </td>
        <td colspan="3">
            <asp:TextBox ID="TextBox2" runat="server" Style="width: 240px;">
            </asp:TextBox>
        </td>
    </tr>
    <tr>
        <td style="text-align: right; width: 69px;">
            Cidade:
        </td>
        <td style="width: 125px;">
            <asp:TextBox ID="TextBox3" runat="server"></asp:TextBox>
        </td>
        <td style="text-align: right; width: 41px;">
            Estado:
        </td>
        <td>
            <asp:DropDownList ID="DropDownList1" runat="server">
                <asp:ListItem>RS</asp:ListItem>
                <asp:ListItem>SC</asp:ListItem>
                <asp:ListItem>RJ</asp:ListItem>
                <asp:ListItem>SP</asp:ListItem>
                <asp:ListItem>PR</asp:ListItem>
            </asp:DropDownList>
        </td>
    </tr>
    <tr>
        <td style="width: 69px; text-align: right; height: 30px;" valign="top">
        </td>

        <td colspan="3" style="height: 30px;">
            <asp:Button ID="Button1" runat="server" Text="OK" />
            <asp:Button ID="Button2" runat="server" Text="Cancelar" />
        </td>
    </tr>
</table>
</fieldset>
</div>
</form>
</body>
</html>

```

Observe o resultado na figura 7.2:

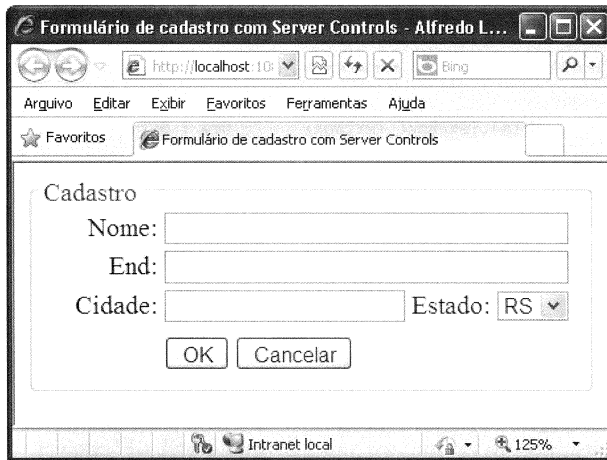


Figura 7.2 – Formulário de cadastro com Server Controls.

O terceiro modo é exclusivo do ASP.NET MVC e usa HTML helpers.

Exemplo:

```
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
    <head runat="server">
        <title>Formulário de cadastro com HTML Helpers</title>
    </head>
    <body>
        @using (Html.BeginForm()) {
            <fieldset style="width: 330px">
                <legend>Cadastro</legend>
                <table style="width: 100%;">
                    <tr>
                        <td style="text-align: right; width: 69px;">
                            @Html.Label("Nome:")
                        </td>
                        <td colspan="3">
                            @Html.TextBox("TextBox1", "", new { @style = "width: 240px;" })
                        </td>
                    </tr>
                </table>
            </fieldset>
        }
```

```

<tr>
  <td style="text-align: right; width: 69px;">
    @Html.Label("End:")
  </td>
  <td colspan="3">
    @Html.TextBox("TextBox2", "", new { @style = "width: 240px;" })
  </td>
</tr>
<tr>
  <td style="text-align: right; width: 69px;">
    @Html.Label("Cidade:")
  </td>
  <td style="width: 125px;">
    @Html.TextBox("TextBox3")
  </td>
  <td style="text-align: right; width: 41px;">
    @Html.Label("Estado:")
  </td>
  <td>
    @Html.DropDownList("DropDownList1", new[] {
      new SelectListItem { Text = "RS", Value = "1" },
      new SelectListItem { Text = "SC", Value = "2" },
      new SelectListItem { Text = "RJ", Value = "3" },
      new SelectListItem { Text = "SP", Value = "4" },
      new SelectListItem { Text = "PR", Value = "5" }
    })
  </td>
</tr>
<tr>
  <td style="width: 69px; text-align: right; height: 30px;" valign="top">
  </td>
  <td colspan="3" style="height: 30px;">
    <input id="Button1" value="OK" type="submit" />
    <input id="Button2" value="Cancelar" type="submit" />
  </td>
</tr>
</table>
</fieldset>
}
</body>
</html>

```

Na figura 73 vemos o formulário no navegador. Repare que todos os 3 formulários são interpretados da mesma forma. Não há diferença no design.

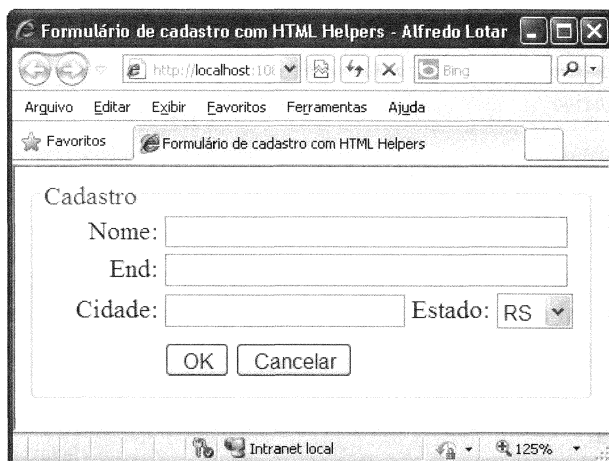


Figura 7.3 – Formulário de cadastro com HTML Helpers.

7.3 Preparando um formulário

Quando usamos HTML helpers, o início e o fim de um formulário são delimitados pelo trecho de código a seguir:

```
@using (Html.BeginForm()) {  
    // Restante do código  
}
```

Se preferir, use o método `EndForm`. O método `BeginForm` marca o início do formulário e `EndForm`, o fim:

```
@{ Html.BeginForm(); }  
    // Restante do código  
@{ Html.EndForm(); }
```

O formulário é postado para o mesmo método de controlador que o invocou. Para postá-lo para outro método de controlador, modifique o método `BeginForm`:

```
@using (Html.BeginForm("Create", "Produtos")) {  
    // Restante do código  
}
```

O navegador interpreta o trecho de código anterior como:

```
<form action="/Produtos/Create" method="post">  
    ... conteúdo do formulário ...  
</form>
```

O método `BeginRouteForm` tem a mesma função do método `BeginForm`.

Exemplo:

```
@using (Html.BeginRouteForm(new { controller = "Produtos", action = "Create" })) {  
    // Restante do código  
}
```

7.4 Exibindo legendas

Os métodos `Label`, `LabelFor` e `LabelForModel` exibem legendas/rótulos para um campo de formulário ou um conteúdo qualquer em uma página, como uma mensagem ao usuário, por exemplo.

7.4.1 Método `Label`

A sintaxe Razor do método `Label` é:

```
@Html.Label(string expression)
```

Exemplo:

```
@Html.Label("Cidade:")
```

Ou:

```
@Html.Label(Model.CategoryName)
```

A linha:

```
@Html.Label("Cidade:")
```

É interpretada no navegador como:

```
<label for="Cidade:">Cidade:</label>
```

Enquanto a linha:

```
@Html.Label(Model.CategoryName)
```

Exibe o conteúdo do campo `CategoryName` e é interpretada como:

```
<label for="Beverages">Beverages</label>
```

Os métodos com o prefixo `Label` assim como os demais métodos abordados nesse capítulo devem ser declarados com o método `BeginForm`:

Exemplo:

```
@{  
    Layout = null;  
}
```

```

<!DOCTYPE html>
<html>
  <head>
    <title>Método Html.Label</title>
  </head>
  <body>
    <div>
      @using (Html.BeginForm()) {
        <fieldset>
          <legend>Html.Label em ação</legend>
          <p>
            @Html.Label("Cidade:")
          </p>
        </fieldset>
      }
    </div>
  </body>
</html>

```

7.4.2 Método LabelFor

Usa expressões lambda para definir o conteúdo exibido na tela.

A sintaxe do método `LabelFor` é:

```
@Html.LabelFor(Expression<Func<TModel, TValue>> expression)
```

Exemplo:

```
@Html.LabelFor(x => x.CategoryName)
```

`CategoryName` é uma propriedade da classe `Category` declarada no topo da página:

```
@model AppCapitulo7.Models.Category
```

7.4.3 Método LabelForModel

Extraí o conteúdo de uma propriedade representada em um modelo.

Exemplo:

```
@Html.LabelForModel(Model.CategoryName)
```

7.5 Caixas de texto

O método `TextBox` e `TextBoxFor` definem os campos de um formulário. Também conhecidos como caixas de texto.

7.5.1 Método TextBox

A sintaxe básica do método `TextBox` requer apenas o nome (identificador) da caixa de texto:

```
@Html.TextBox("txtNome")
```

O navegador interpreta a linha como:

```
<input id="txtNome" name="txtNome" type="text" value=""/>
```

Você pode também especificar um valor inicial:

```
@Html.TextBox("txtNome", "Digite o seu nome aqui.")
```

A quantidade de caracteres do campo:

```
@Html.TextBox("txtNome", "Digite o seu nome aqui.", new { @size = 50 })
```

No terceiro parâmetro, outros atributos HTML podem ser declarados:

```
@Html.TextBox(idControle, Texto, atributosHTML)
```

Como a formatação usada, definida via seletores CSS:

```
@Html.TextBox("txtNome", "Digite o seu nome aqui.", new { @Class = "input-validation-error" })
```

Múltiplos atributos HTML devem ser separados por vírgula:

```
@Html.TextBox("txtNome", "Digite o seu nome aqui.", new { @size = 50,
    @Class = "input-validation-error" })
```

Exemplo:

```
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
    <head>
        <title>Método Html.TextBox</title>
    </head>
    <body>
        <div>
            @using (Html.BeginForm()) {
                <fieldset>
                    <legend>Html.TextBox em ação</legend>
                    <p>
                        @Html.TextBox("txtNome", "Digite o seu nome aqui.",
                            new { @size = 50, @Class = "input-validation-error" })
                    </p>
                </fieldset>
            }
        </div>
    </body>
</html>
```



```

    </div>
  </body>
</html>

```

A figura 7.4 ilustra o funcionamento do método `TextBox`:

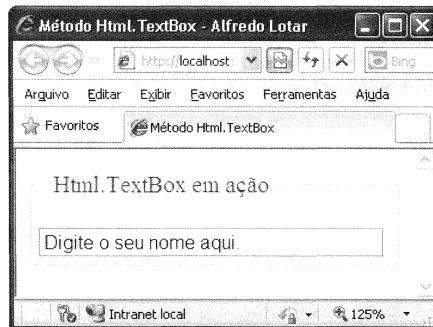


Figura 7.4 – Método `Html.TextBox` em ação.

7.5.2 Método `TextBoxFor`

Define o nome do campo e também o conteúdo por intermédio de expressões lambda. Útil quando você precisa preencher os campos de um formulário de atualização com as informações atuais do registro.

Exemplo:

```

@Html.TextBoxFor(x => x.CategoryName)
@Html.TextBoxFor(x => x.CategoryName, new { @Class = "input-validation-error" })

```

`CategoryName` é uma propriedade da classe `Category`.

7.5.3 Método `TextArea`

O método `TextArea` apresenta o conteúdo de uma caixa de texto em múltiplas linhas.

Exemplo:

```

@Html.TextArea("txtNome")
@Html.TextArea("txtNome", "Digite o seu nome aqui.")
@Html.TextArea("txtNome", "Digite o seu nome aqui.", new { @Class = "textarea" })

```

O navegador interpreta as linhas como:

```

<textarea cols="20" id="txtNome" name="txtNome" rows="2"></textarea>
<textarea cols="20" id="txtNome" name="txtNome" rows="2">Digite o seu nome aqui.
</textarea>
<textarea Class="textarea" cols="20" id="txtNome" name="txtNome" rows="2">
  Digite o seu nome aqui.
</textarea>

```

A figura 7.5 mostra o exemplo em ação:

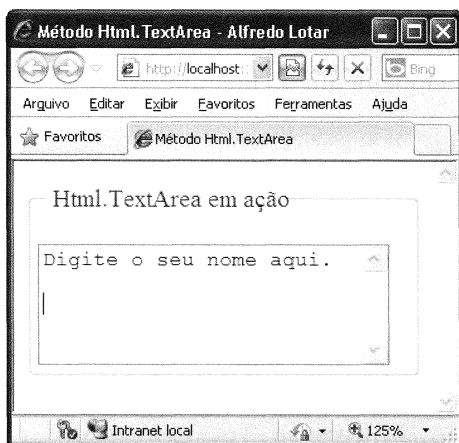


Figura 7.5 – Método *Html.TextArea* em ação.

7.5.4 Método *TextAreaFor*

Requer expressões lambda.

Exemplo:

```
@Html.TextAreaFor(x => x.CategoryName)
```

```
@Html.TextAreaFor(x => x.CategoryName, new { @Class = "textarea" })
```

Ambas as linhas são interpretadas no navegador como:

```
<textarea cols="20" id="CategoryName" name="CategoryName" rows="2">Beverages</textarea>
```

```
<textarea Class="textarea" cols="20" id="CategoryName" name="CategoryName" rows="2">Beverages</textarea>
```

E retornam o conteúdo apresentado na figura 7.6:

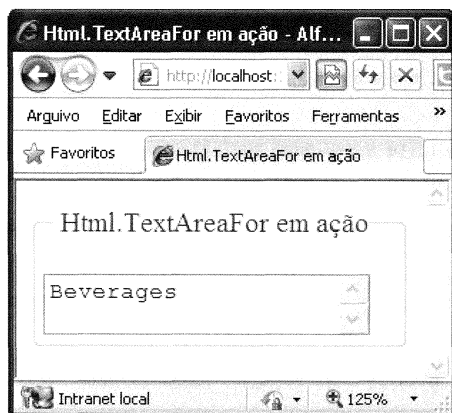


Figura 7.6 – Método *Html.TextAreaFor* em ação.

7.5.5 Métodos Hidden e HiddenFor

Criam campos ocultos em um formulário. A sintaxe é a mesma do método `TextBox`.

Exemplo:

```
@Html.Hidden("txtNome")
@Html.Hidden("txtNome", "texto do campo oculto.")
@Html.Hidden("txtNome", "texto do campo oculto.", new { @Class = "SeletorCSS" })
@Html.HiddenFor(x => x.CategoryName)
@Html.HiddenFor(x => x.CategoryName, new { @Class = "SeletorCSS" })
```

7.5.6 Método Password e PasswordFor

Apresenta o conteúdo de um campo de formulário com uma máscara ocultando os caracteres digitados pelo usuário. Cria as famosas caixas de texto para senhas.

Exemplo:

```
@Html.Password("txtSenha")
@Html.Password("txtSenha", "senhaaqui")
@Html.Password("txtSenha", "senhaaqui", new { @Class = "SeletorCSS" })
@Html.PasswordFor(x => x.CategoryName)
@Html.PasswordFor(x => x.CategoryName, new { @Class = "SeletorCSS" })
```

Observe a figura 7.7:



Figura 7.7 – *Html.Password em ação.*

7.6 Caixas de seleção

O método `CheckBox` cria uma caixa de seleção e permite que um usuário selecione múltiplos elementos de uma lista. Você, por exemplo, pode fazer a seguinte pergunta a um usuário. Quais são suas cores preferidas? E, em seguida, criar um método `CheckBox` para cada cor. O usuário pode selecionar uma ou mais cores da lista.

Um item com uma marca de seleção significa que o item está ativo.

A sintaxe do método `CheckBox` é a seguinte:

```
@Html.CheckBox(idControle, AtivoSim/Não, atributosHTML)
```

Exemplo:

```
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
    <head>
        <title>Método Html.CheckBox</title>
        <link href="@Url.Content("~/Content/Site.css")" rel="stylesheet" type="text/css"/>
    </head>
    <body>
        <div>
            @using (Html.BeginForm()) {
                <fieldset>
                    <legend>Quais são suas cores preferidas?</legend>
                    <p>
                        Azul @Html.CheckBox("chkAzul")
                        Verde @Html.CheckBox("chkVerde", true)
                        Vermelho @Html.CheckBox("chkVermelho")
                        Branco @Html.CheckBox("chkBranco")
                        Outra cor @Html.CheckBox("chkOutra")
                    </p>
                </fieldset>
            }
        </div>
    </body>
</html>
```

A figura 7.8 ilustra o exemplo em ação:

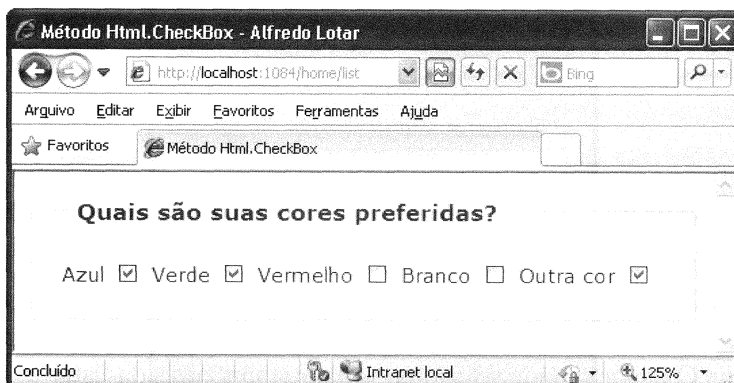


Figura 7.8 – Caixas de seleção com o método `Html.CheckBox`.

7.7 Botões de rádio

Chamados de botões de rádio ou botões de opções, existem em um grupo no qual somente um deles pode ser selecionado pelo usuário.

`Html.RadioButton` exibe um botão de rádio na tela:

```
@Html.RadioButton("rndEstadoCivil", 1, true) Solteiro (a)
```

O navegador interpreta a linha como:

```
<input checked="checked" id="rndEstadoCivil" name="rndEstadoCivil" type="radio" value="1" />
  Solteiro (a)
```

Obs.: para agrupar botões de rádio defina todos com o mesmo nome.

Por exemplo, faça a seguinte pergunta: Qual seu estado civil?

```
@Html.RadioButton("rndEstadoCivil", 1, true) Solteiro (a)
```

```
@Html.RadioButton("rndEstadoCivil", 2) Casado (a)
```

```
@Html.RadioButton("rndEstadoCivil", 3) Separado (a)
```

```
@Html.RadioButton("rndEstadoCivil", 4) Outro (a)
```

Exemplo:

```
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
    <head>
        <title>Método Html.RadioButton</title>
        <link href="@Url.Content("~/Content/Site.css")" rel="stylesheet" type="text/css"/>
    </head>
    <body>
        <div>
            @using (Html.BeginForm()) {
                <fieldset>
                    <legend>Qual seu estado civil?</legend>
                    <p>
                        @Html.RadioButton("rndEstadoCivil", 1, true) Solteiro (a)
                        @Html.RadioButton("rndEstadoCivil", 2) Casado (a)
                        @Html.RadioButton("rndEstadoCivil", 3) Separado (a)
                        @Html.RadioButton("rndEstadoCivil", 4) Outro (a)
                    </p>
                </fieldset>
            }
        </div>
    </body>
</html>
```

Na figura 7.9 vemos o formulário com os botões de rádio:

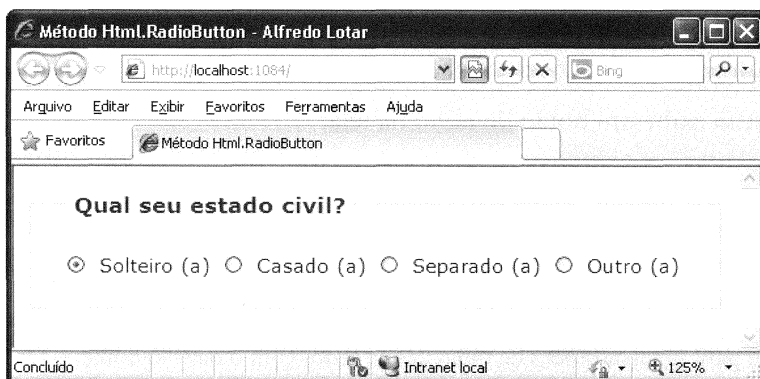


Figura 7.9 – Botões de rádio com o método `Html.RadioButton`.

7.8 Caixa de combinação

Uma caixa de combinação contém uma lista de itens predefinida. Geralmente, somente o item-padrão é visível. Os demais itens ficam ocultos até que o usuário clique no botão **Drop-down**.

O método `DropDownList` ou `DropDownListFor` exibe uma caixa de combinação.

As principais sintaxes do método `DropDownList` são:

```
@Html.DropDownList(idControle, IEnumerable<SelectListItem> selectList)
@Html.DropDownList(idControle, IEnumerable<SelectListItem> selectList,
    string TextoItemEmBranco, atributosHTML)
```

Exemplo:

```
@using (Html.BeginForm()) {
    <fieldset>
        <legend>Selecione um País</legend>
        <p>
            Países @Html.DropDownList("dboPaíses", new[] {
                new SelectListItem { Text = "Brasil", Value = "1" },
                new SelectListItem { Text = "Alemanha", Value = "2" },
                new SelectListItem { Text = "Austrália", Value = "3" },
                new SelectListItem { Text = "Uruguai", Value = "4" },
                new SelectListItem { Text = "Japão", Value = "5" }
            }, "Selecione")
        </p>
    </fieldset>
}
```

Observe a saída na figura 7.10:



Figura 7.10 – Caixa de combinação com o método `Html.DropDownList`.

Se necessário, carregue os itens de uma caixa de combinação a partir de um método público de controlador.

Exemplo:

```
public class HomeController : Controller {
    public ActionResult List() {
        var lst = new List<string> {
            "Brasil", "Alemanha", "Austrália", "Uruguai", "Japão"
        };
        ViewBag.Paises = new SelectList(lst);
        return View();
    }
}
```

A propriedade `dynamic Paises` contém a lista de itens:

```
ViewBag.Paises = new SelectList(lst);
```

No view associado ao método `List` exibimos os itens:

```
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
    <head>
        <title>Método Html.DropDownList</title>
    </head>
    <body>
        <div>
            @using (Html.BeginForm()) {
```

```

        @Html.DropDownList("Países")
    }
</div>
</body>
</html>

```

A propriedade Países:

```
ViewBag.Paises = new SelectList(lst);
```

É passada ao método DropDownList:

```
@Html.DropDownList("Países")
```

Obs.: no exemplo, usamos um objeto List, mas você pode usar outra origem de dados qualquer, como um array, banco de dados, um documento XML etc.

7.8.1 Método DropDownListFor

O método DropDownListFor usa expressões lambda e é útil em um formulário de atualização:

```

Produto: @Html.DropDownListFor(x => x.Discontinued, new[]
    { new SelectListItem() { Text = "Ativo", Value = "true" },
      new SelectListItem() { Text = "Inativo", Value = "false" } })

```

Exemplo:

```

@model AppCapitulo7.Models.Product
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
    <head>
        <title>Método Html.DropDownListFor</title>
    </head>
    <body>
        <div>
            @using (Html.BeginForm()) {
                <p>
                    Produto: @Html.DropDownListFor(x => x.Discontinued, new[] { new SelectListItem()
                        { Text = "Ativo", Value = "true" }, new SelectListItem()
                        { Text = "Inativo", Value = "false" } })
                </p>
            }
        </div>
    </body>
</html>

```


7.9 Caixa de listagem

Uma caixa de listagem exibe itens com base no número de linhas visíveis permitidas. Você pode selecionar um ou mais itens. O método `ListBox` ou `ListBoxFor` cria uma caixa de listagem na tela.

A sintaxe do método `ListBox` é a seguinte:

```
@Html.ListBox(idControle)
@Html.ListBox(idControle, IEnumerable<SelectListItem> selectList)
@Html.ListBox(idControle, IEnumerable<SelectListItem> selectList, atributosHTML)
```

Exemplo:

```
@Html.ListBox("lstBebidas", new MultiSelectList(new[] { "Vinho", "Cerveja" }))
@Html.ListBox("lstPaises", new[] {
    new SelectListItem { Text = "Brasil", Value = "1" },
    new SelectListItem { Text = "Alemanha", Value = "2" },
    new SelectListItem { Text = "Austrália", Value = "3" },
    new SelectListItem { Text = "Uruguai", Value = "4" },
    new SelectListItem { Text = "Japão", Value = "5" }}
)
```

No navegador é interpretado como:

```
<select id="lstBebidas" multiple="multiple" name="lstBebidas">
    <option>Vinho</option>
    <option>Cerveja</option>
</select>
<select id="lstPaises" multiple="multiple" name="lstPaises">
    <option value="1">Brasil</option>
    <option value="2">Alemanha</option>
    <option value="3">Australia</option>
    <option value="4">Uruguai</option>
    <option value="5">Japão</option>
</select>
```

Os itens da caixa de listagem podem ser carregados também a partir de um método de controlador:

```
public class HomeController : Controller {
    public ActionResult List() {
        var lst = new List<string> {
            "Brasil", "Alemanha", "Austrália", "Uruguai", "Japão"
        };
        ViewBag.Paises = new SelectList(lst);
        return View();
    }
}
```

No view temos:

```
@using (Html.BeginForm()) {  
    @Html.ListBox("Países")  
}
```

Exemplo:

```
@{  
    Layout = null;  
}  
<!DOCTYPE html>  
<html>  
    <head>  
        <title>Método Html.ListBox</title>  
    </head>  
    <body>  
        <div>  
            <fieldset>  
                <legend>Selecione um País</legend>  
                <p>  
                    @using (Html.BeginForm()) {  
                        @Html.ListBox("Países")  
                    }  
                </p>  
            </fieldset>  
        </div>  
    </body>  
</html>
```

A figura 7.11 mostra uma caixa de listagem com itens:



Figura 7.11 – Caixa de listagem com o método `Html.ListBox`.

7.9.1 Método ListBoxFor

Semelhante ao método `ListBox`, mas usa expressões lambda. Exemplo:

```
@Html.ListBoxFor(x=>x.ProductName, new MultiSelectList(new[] { "Vinho", "Cerveja" }))
```

7.10 Métodos de edição

Os métodos de edição do ASP.NET MVC são simples e fáceis de usar. Por exemplo, com algumas linhas de código você tem um formulário de atualização completo.

7.10.1 Método EditorForModel

O método `EditorForModel` é ótimo para editar dados rapidamente. Por exemplo, se você tem um método `Edit` em um controlador e seleciona um registro específico por meio de instruções SQL ou LINQ, para editar as informações desse registro, basta associar um método e a classe usada a um view e declarar o método `EditorForModel`.

Por exemplo, no controlador, temos dois métodos `Edit`:

```
public ActionResult Edit(int? id) {
    try {
        if (!id.HasValue)
            return View("NotFound");
        var model = db.Categories.Where(c => c.CategoryID == id).FirstOrDefault();
        if (model == null)
            return View("NotFound");
        else
            return View(model);
    }
    finally {
        if (db != null) db.Dispose();
    }
}

[HttpPost]
public ActionResult Edit(int? id, FormCollection collection) {
    try {
        if (!id.HasValue)
            return View("NotFound");

        var model = db.Categories.Where(c => c.CategoryID == id).FirstOrDefault();
        if (model == null) {
            return View("NotFound");
        }
    }
}
```

```

        else {
            this.TryUpdateModel(model);
            db.SaveChanges();
        }
        return RedirectToAction("Index");
    }
    catch {
        return View();
    }
    finally {
        if (db != null) db.Dispose();
    }
}

```

E, no view `Edit.aspx`, declaramos o método `EditorForModel`:

```

@model AppCapitulo7.Models.Category
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
    <head>
        <title>Exemplo simples com o método Html.EditorForModel</title>
    </head>
    <body>
        <div>
            @using (Html.BeginForm()) {
                @Html.EditorForModel()
                <input id="edit" type="submit" value="Editar"/>
            }
        </div>
    </body>
</html>

```

Conforme observamos na figura 7.12, o formulário é carregado com todas as informações do registro. Ou seja, pronto para edição. Não é preciso criar vários controles `Label` e `TextBox`.

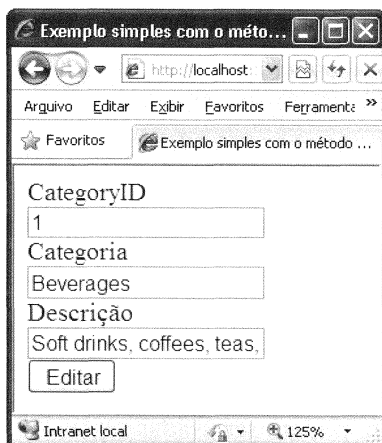


Figura 7.12 – Formulário de edição exibido com o método `Html.EditorForModel`.

7.10.2 Método Editor

Retorna um elemento HTML input para a propriedade representada pela expressão.

`@Html.Editor(expressão)`

Por exemplo:

`@Html.Editor(Model.CategoryName)`

Ou:

`@Html.Editor("CategoryName")`

Retorna:

```
<input class="text-box single-line" id="Beverages" name="Beverages" type="text" value="" />
```

7.10.3 Método EditorFor

Usa expressões lambda. É semelhante ao método `Editor`:

`@Html.EditorFor(x => x.CategoryName)`

7.11 Exibindo Texto

No ASP.NET MVC temos diversos métodos que retornam o conteúdo de uma propriedade como uma string.

Por exemplo, o método `DisplayTextFor` é semelhante ao método `EditorForModel`, mas exibe o conteúdo como uma lista de strings e não em um formulário.

As linhas a seguir exibem o conteúdo de todas as propriedades da classe associada ao view:

```
@using (Html.BeginForm()) {  
    @Html.DisplayForModel()  
}
```

Se preferir, exiba o conteúdo de uma propriedade específica:

```
@Html.Display("CategoryName")
```

Ou:

```
@Html.DisplayText("CategoryName")
```

Expressões lambda também podem ser usadas:

```
@Html.DisplayFor(x => x.CategoryName)
```

Ou:

```
@Html.DisplayTextFor(x => x.CategoryName)
```

7.12 Ativando e desativando elementos HTML

Por motivos de segurança, antes de exibir uma informação qualquer no navegador, devemos sempre desativar caracteres HTML inseridos propositalmente ou não, no campo de formulário ou no banco de dados. Os métodos abordados a seguir nos auxiliam nessa tarefa.

7.12.1 Método Encode

O método `Encode` é semelhante ao método `HtmlEncode` do objeto `Server`.

Exemplo:

```
@Html.Encode("<b>Alfredo Lotar</b>")
```

Retorna:

```
&lt;b>Alfredo Lotar&lt;/b>
```

7.12.2 Método AttributeEncode

É um pouco diferente do método `Encode`, pois codifica somente aspas duplas, o caractere `&` e também `<`. Exemplo:

```
@Html.AttributeEncode("<b>Alfredo Lotar</b>")
```

Retorna:

```
&lt;b>Alfredo Lotar&lt;/b>
```

7.12.3 Método Raw

Diferente de versões anteriores do ASP.NET. O ASP.NET MVC codifica o texto enviado para o navegador, automaticamente.

Quando a string:

```
"<b>Alfredo Lotar</b>"
```

É definida em um método de controlador:

```
public ActionResult Index() {  
    ViewBag.Nome = "<b>Alfredo Lotar</b>";  
    return View();  
}
```

E declarada em um view:

```
<p>  
    @ViewBag.Nome  
</p>
```

É exibida em um navegador como:

```
<b>Alfredo Lotar</b>
```

Se você deseja que o código HTML seja interpretado, use o método `Raw`:

```
<p>  
    @Html.Raw(ViewBag.Nome)  
</p>
```

No navegador temos:

Alfredo Lotar

Em negrito.

7.13 Formulário de inserção de dados

Para inserir informações em um banco de dados defina no controlador dois métodos `Create`. O primeiro exibe o formulário em branco:

```
public ActionResult Create() {  
    return View();  
}
```

E o segundo envia as informações para o banco de dados:

```
[HttpPost]  
public ActionResult Create(Category entidade) {  
    try {  
        if (!ModelState.IsValid) {  
            ModelState.AddModelError("",  
                "Ocorreram erros no formulário. Por favor, corrija os erros e tente novamente.");  
        }  
    }  
}
```

```
        return View();
    }
    else {
        db.AddToCategories(entidade);
        db.SaveChanges();
        return RedirectToAction("Index");
    }
}
}
catch(System.Data.EntitySqlException) {
    ModelState.AddModelError("", "Erro EntitySqlException.");
    return View();
}
catch (System.Data.EntityCommandExecutionException) {
    ModelState.AddModelError("", "Erro EntityCommandExecutionException.");
    return View();
}
finally {
    if (db != null) db.Dispose();
}
}
```

Obs.: no exemplo, nomeamos o método como Create, mas você pode usar qualquer outro nome, como: Inserir, Adicionar, Acrescentar, Add etc.

O modo mais fácil de se criar um formulário de inserção de dados é por intermédio da caixa de diálogo **Add View**. Clique com o botão direito do mouse no método Create, em seguida, selecione **Add View...**. Observe na figura 7.13 a caixa de diálogo **Add View** configurada para adicionar ao projeto o template Create.

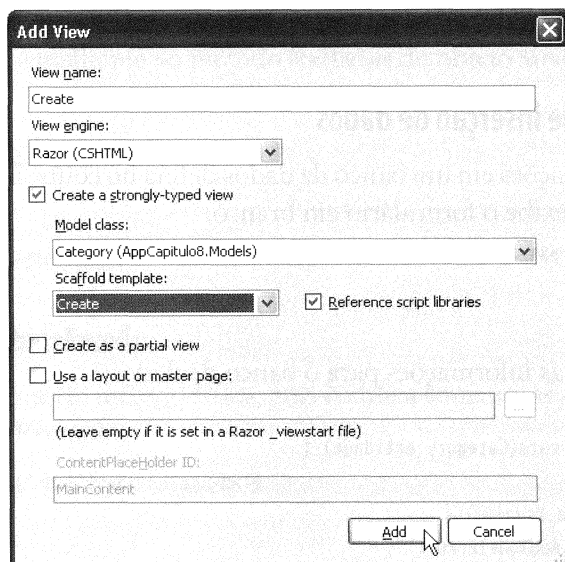


Figura 7.13 – Caixa de diálogo Add View.

Repare o código HTML e os HTML helpers do view `Create.cshtml`. É importante que o ID dos campos do formulário seja igual ao nome dos parâmetros do método `Create` ou igual ao nome das propriedades da classe associada `Category`:

```
@model AppCapitulo7.Models.Category
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
    <head>
        <title>Create</title>
        <script src="@Url.Content("~/Scripts/jquery-1.4.4.min.js")" type="text/javascript">
        </script>
        <script src="@Url.Content("~/Scripts/jquery.validate.min.js")" type="text/javascript">
        </script>
        <script src="@Url.Content("~/Scripts/jquery.validate.unobtrusive.min.js")"
            type="text/javascript"></script>
        <link href="@Url.Content("~/Content/Site.css")" rel="stylesheet" type="text/css"/>
    </head>
    <body>
        @using (Html.BeginForm()) {
            @Html.ValidationSummary(true)
            <fieldset>
                <legend>Nova categoria</legend>
                <div class="editor-label">
                    @Html.LabelFor(model => model.CategoryName)
                </div>
                <div class="editor-field">
                    @Html.TextBoxFor(model => model.CategoryName, new { @style = "width: 250px;" })
                    @Html.ValidationMessageFor(model => model.CategoryName)
                </div>
                <div class="editor-label">
                    @Html.LabelFor(model => model.Description)
                </div>
                <div class="editor-field">
                    @Html.TextAreaFor(model => model.Description, new { @style = "width: 250px;" })
                    @Html.ValidationMessageFor(model => model.Description)
                </div>
                <p>
                    <input type="submit" value="Create" />
                </p>
            </fieldset>
        }
    </body>
</html>
```

```

<div>
    @Html.ActionLink("Página inicial", "Index")
</div>
</body>
</html>

```

A figura 7.14 mostra o formulário de inserção em ação:

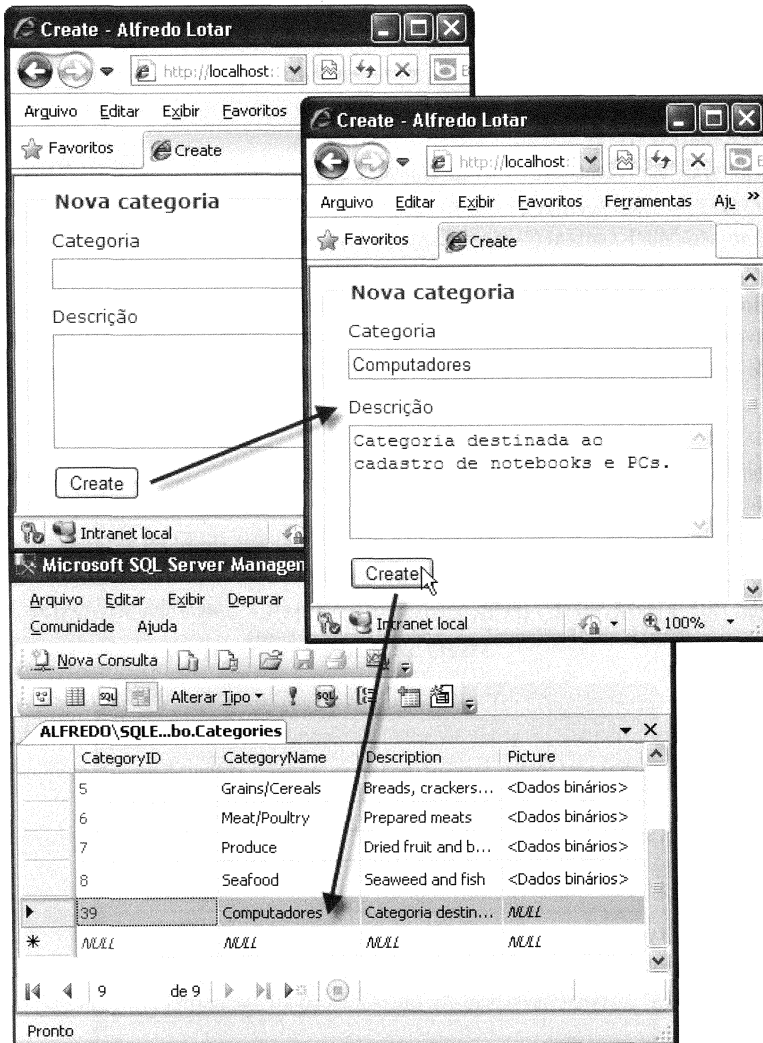


Figura 7.14 – Inserindo uma nova categoria.

7.14 Formulário de atualização de dados

A atualização de informações é semelhante à inserção. Primeiramente, carregamos a lista de categorias:

```
private NorthwindBD db = new NorthwindBD();
public ActionResult Index() {
    try {
        var model = db.Categories.AsParallel().ToList();
        return View(model);
    }
    finally {
        if (db != null) db.Dispose();
    }
}
```

O view `Index.cshtml` exibe as informações no navegador:

```
@model IEnumerable<AppCapitulo7.Models.Category>
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
    <head>
        <title>Lista de categorias</title>
    </head>
    <body>
        <div>
            <ul>
                @foreach (var item in Model) {
                    <li>
                        @Html.ActionLink(item.CategoryName, "Edit", new { id = item.CategoryID })
                    </li>
                }
            </ul>
            @Html.ActionLink("Create", "create")
        </div>
    </body>
</html>
```

Em seguida, como já abordado neste livro, são necessários dois métodos `Edit`. O primeiro carrega os dados atuais do registro.

```
public ActionResult Edit(int? id) {
    try {
        if (!id.HasValue)
            return View("NotFound");
        var model = db.Categories.Where(c => c.CategoryID == id).FirstOrDefault();
        if (model == null)
            return View("NotFound");
        else
            return View(model);
    }
    finally {
        if (db != null) db.Dispose();
    }
}
```

E o segundo método `Edit` envia as alterações para o banco de dados:

```
[HttpPost]
public ActionResult Edit(Category entidade) {
    try {
        var model = db.Categories.Where(c => c.CategoryID == entidade.CategoryID).
            FirstOrDefault();
        if (model == null) {
            return View("NotFound");
        }
        else {
            db.ApplyCurrentValues(model.EntityKey.EntitySetName, entidade);
            db.SaveChanges();
            return RedirectToAction("Index");
        }
    }
    finally {
        if (db != null) db.Dispose();
    }
}
```

O passo seguinte é a criação do view com o formulário de atualização. Na caixa de diálogo **Add View**, definimos o nome do view, a classe associada e o template `Edit`. Observe a figura 7.15:

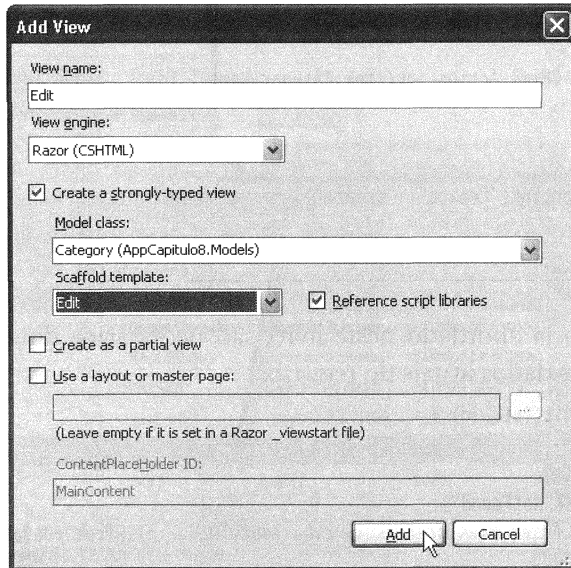


Figura 7.15 – Caixa de diálogo *Add View*.

O código do template `Edit` vemos a seguir:

```
@model AppCapitulo7.Models.Category
@{
    Layout = null;
}
```

```

<!DOCTYPE html>
<html>
  <head>
    <title>Edit</title>
    <script src="@Url.Content("~/Scripts/jquery-1.4.4.min.js")" type="text/javascript">
    </script>
    <script src="@Url.Content("~/Scripts/jquery.validate.min.js")" type="text/javascript">
    </script>
    <script src="@Url.Content("~/Scripts/jquery.validate.unobtrusive.min.js")"
      type="text/javascript">
    </script>
    <link href="@Url.Content("~/Content/Site.css")" rel="stylesheet" type="text/css"/>
  </head>
  <body>
    @using (Html.BeginForm("Edit", "Home")) {
      @Html.ValidationSummary(true)
      <fieldset>
        <legend>Editar categoria</legend>
        @Html.HiddenFor(model => model.CategoryID)
        <div class="editor-label">
          @Html.LabelFor(model => model.CategoryName)
        </div>
        <div class="editor-field">
          @Html.TextBoxFor(model => model.CategoryName, new { @style = "width: 250px;" })
          @Html.ValidationMessageFor(model => model.CategoryName)
        </div>
        <div class="editor-label">
          @Html.LabelFor(model => model.Description)
        </div>
        <div class="editor-field">
          @Html.TextAreaFor(model => model.Description, new { @style = "width: 250px;" })
          @Html.ValidationMessageFor(model => model.Description)
        </div>
        <p>
          <input type="submit" value="Save" />
        </p>
      </fieldset>
    }
    <div>
      @Html.ActionLink("Página inicial", "Index")
    </div>
  </body>
</html>

```

A figura 7.16 ilustra o formulário de atualização em ação:

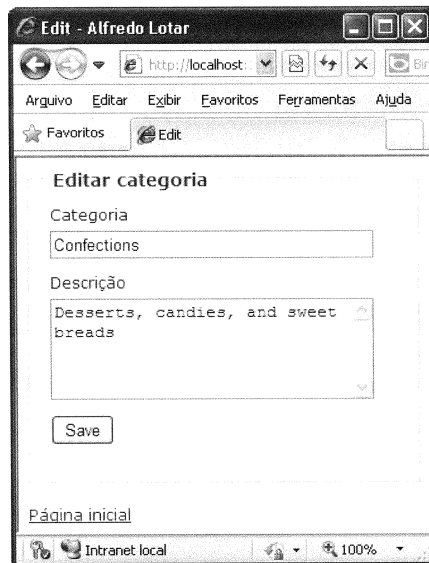


Figura 7.16 - Editando uma categoria.

7.15 Criando HTML Helpers

Você pode desenvolver seu próprio HTML helper ou usar HTML helpers desenvolvidos por terceiros, como empresas ou desenvolvedores independentes.

Um HTML helper pode ser simples, como o desenvolvido a seguir, no qual criamos um botão Submit.

Criar um novo HTML helper é fácil. Você cria uma nova classe static e define uma namespace. O código que faz a coisa acontecer, digamos assim. É inserido em um extension method, ou seja, é um método static, e o primeiro parâmetro é precedido do modificador this. Significa que estende o tipo `HtmlHelper`.

Para criar um HTML helper na aplicação `AppCapitulo7`, siga os passos descritos a seguir:

Crie um diretório separado, conforme a figura 7.17.

Crie também a classe `Helpers` no arquivo `HtmlHelpers.cs` e insira o código a seguir:

```
using System.Web.Mvc;

namespace AlfredoLotar.Livro.AspnetMvc {
    public static class Helpers {
        public static MvcHtmlString Submit(this HtmlHelper helper, string text) {
            string str = string.Format("<input id=\"Submit1\" type=\"submit\" value=\"{0}\"/>", text);
            return new MvcHtmlString(str);
        }
    }
}
```

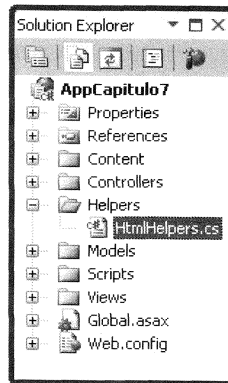


Figura 7.17 – Diretório Helpers e o arquivo HtmlHelpers.cs.

A classe `MvcHtmlString` faz com que as tags HTML sejam interpretadas pelo navegador:

```
return new MvcHtmlString(str);
```

A linha anterior pode ser reescrita como:

```
return MvcHtmlString.Create(str);
```

Para exibir texto simples, altere o tipo de retorno e elimine a classe `MvcHtmlString` do código:

```
public static string Exemplo(this HtmlHelper helper, string text) {
    return text;
}
```

No view importe a namespace `AlfredoLotar.Livro.AspNetMvc`:

```
<%@ Import Namespace="AlfredoLotar.Livro.AspNetMvc" %>
```

E invoque o método `Submit` como se fosse um HTML helper nativo:

```
<%:Html.Submit("Enviar")%>
```

Exemplo:

```
<%@ Page Language="C#" Inherits="System.Web.Mvc.ViewPage<dynamic>" %>
<%@ Import Namespace="AlfredoLotar.Livro.AspNetMvc" %>
<!DOCTYPE html>
<html>
    <head runat="server">
        <title>Index</title>
    </head>
    <body>
        <div>
            <% using (Html.BeginForm()) {%>
                <fieldset>
                    <legend>Botão submit de exemplo</legend>
```

```

        <%=Html.Submit("Enviar")%>
    </fieldset>
    <% } %>
</div>
</body>
</html>

```

Agora, se você usa a sintaxe Razor, a coisa é um pouco diferente. A palavra-chave `using` importa a namespace para o view:

```
@using AlfredoLotar.Livro.AspNetMvc
```

O método `Submit` pode ser invocado como qualquer outro HTML helper:

```

@using (Html.BeginForm()) {
    <fieldset>
        <legend>Botão submit de exemplo</legend>
        <p>
            @Html.Submit("Enviar")
        </p>
    </fieldset>
}

```

7.15.1 HTML helpers inline

A sintaxe Razor permite HTML helpers inline, ou seja, o código do método é inserido diretamente no view por intermédio do bloco `@helper`:

```

@helper Submit(string text) {
    MvcHtmlString ctr = new MvcHtmlString(String.Format("<input id=\"Submit1\"
        type=\"submit\" value=\"{0}\"/>", text));
    @ctr;
}

```

Exemplo:

```

@{
    Layout = null;
}
<!DOCTYPE html>
<html>
    <head>
        <title>Botão submit de exemplo</title>
    </head>
    <body>
        @helper Submit(string text) {
            MvcHtmlString ctr = new MvcHtmlString(String.Format("<input id=\"Submit1\"
                type=\"submit\" value=\"{0}\"/>", text));
            @ctr;
        }
    </body>
</html>

```



```

    }
    <div>
        @using (Html.BeginForm()) {
            <fieldset>
                <legend>Botão submit de exemplo</legend>
                <p>
                    @Submit("Enviar")
                </p>
            </fieldset>
        }
    </div>
</body>
</html>

```

7.15.2 Classe TagBuilder

Classe usada por HTML helpers para criar elementos HTML. Neste tópico, empregamos a classe `TagBuilder` para desenvolver um HTML helper e exibir a tag HTML `<img>`.

Primeiro passo é criar uma instância da classe `TagBuilder` e definir o nome da tag que deve ser criada:

```
TagBuilder tb = new TagBuilder("img");
```

Em seguida, definimos o identificador com o método `GenerateId`:

```
tb.GenerateId(id);
```

As demais propriedades são criadas com o método `MergeAttribute` e `MergeAttributes`:

```
tb.MergeAttribute("src", url);
tb.MergeAttribute("alt", alternateText);
tb.MergeAttributes(new RouteValueDictionary(htmlAttributes));
```

Para definir um seletor CSS para a tag, use o método `AddCssClass`:

```
tb.AddCssClass("seletorCSS");
```

O método `ToString` retorna a tag criada. Você pode criar uma tag normal, uma tag inicial, uma tag de fechamento, uma tag vazia ou que contém apenas atributos:

```
tb.ToString(TagRenderMode.SelfClosing);
```

Exemplo:

```

using System.Web.Mvc;
using System.Web.Routing;
namespace AlfredoLotar.Livro.AspNetMvc {
    public static class Helpers {
        public static MvcHtmlString Image(this HtmlHelper helper, string id, string url,
            string alternateText) {

```

```

        return Image(helper, id, url, alternateText, null);
    }

    public static MvcHtmlString Image(this HtmlHelper helper, string id, string url,
        string alternateText, object htmlAttributes) {
        TagBuilder tb = new TagBuilder("img");

        tb.GenerateId(id);

        tb.MergeAttribute("src", url);
        tb.MergeAttribute("alt", alternateText);
        tb.MergeAttributes(new RouteValueDictionary(htmlAttributes));

        return MvcHtmlString.Create(tb.ToString(TagRenderMode.SelfClosing));
    }
}

```

No código anterior temos a opção de incluir a classe `UrlHelper` e formatar a URL e o texto que descreve a imagem. Inclua a linha:

```
UrlHelper _urlHelper = new UrlHelper(helper.ViewContext.RequestContext);
```

E formate a URL:

```
tb.MergeAttribute("src", _urlHelper.Content(url));
```

E o texto:

```
tb.MergeAttribute("alt", _urlHelper.Encode(alternateText));
```

Após a criação do HTML helper, você pode usá-lo em um view. Exemplo:

```

@using AlfredoLotar.Livro.AspNetMvc
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
    <head>
        <title>HTML helper Image</title>
    </head>
    <body>
        <div>
            @Html.Image("idImage", Href("~/Content/imagem.jpg"), "Descrição")
            @Html.Image("idImag", Href("~/Content/imagem.jpg"), "Descrição", new {border = "3" })
        </div>
    </body>
</html>

```

A saída vemos na figura 7.18:

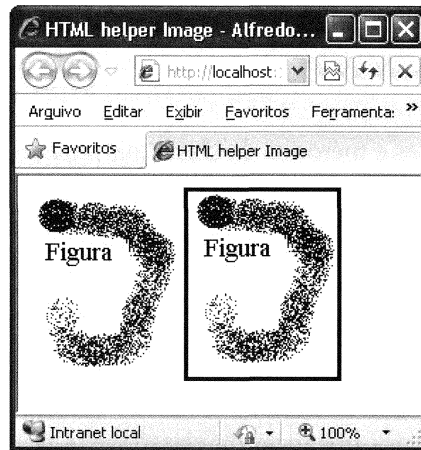


Figura 7.18 - HTML helper Image.

No exemplo anterior usamos a sintaxe Razor. Em um view ASPX, importe a namespace `AlfredoLotar.Livro.AspNetMvc`:

```
<%@ Import Namespace="AlfredoLotar.Livro.AspNetMvc" %>
```

Na linha:

```
@Html.Image("idImage", Href("~/Content/imagem.jpg"), "Descrição")
```

Substitua o método `Href` por `ResolveUrl`:

```
<%:Html.Image("idImag", ResolveUrl("~/Content/imagem.jpg"), "Descrição")%>
```

Exemplo:

```
<%@ Page Language="C#" Inherits="System.Web.Mvc.ViewPage<dynamic>" %>
<%@ Import Namespace="AlfredoLotar.Livro.AspNetMvc" %>
<!DOCTYPE html>
<html>
  <head>
    <title>HTML helper Image</title>
  </head>
  <body>
    <div>
      <%:Html.Image("idImag", ResolveUrl("~/Content/imagem.jpg"), "Descrição")%>
    </div>
  </body>
</html>
```

7.15.3 Classe `HtmlTextWriter`

Se você precisa criar um HTML helper que exibe uma tag HTML que contém outras tags-filha aninhadas, por exemplo, a tag `object`, use a classe `HtmlTextWriter` em vez da classe `TagBuilder`.

As principais propriedades e métodos da classe `HtmlTextWriter` são:

Propriedade/método	Descrição
<code>Indent</code>	Define a endentação do documento.
<code>AddAttribute</code>	Define o nome e o valor do atributo.
<code>RenderBeginTag</code>	Exibe a tag de abertura. O nome é definido por intermédio da enumeração <code>HtmlTextWriterTag</code> .
<code>RenderEndTag</code>	Exibe a tag de fechamento.

Quando você cria uma nova instância da classe `HtmlTextWriter`, passe uma instância da classe `StringWriter`:

```
using (HtmlTextWriter writer = new HtmlTextWriter(new StringWriter()))
```

Exemplo:

```
public static MvcHtmlString ShowFlash(this HtmlHelper helper, string FlashFile, string width,
    string height) {
    using (HtmlTextWriter writer = new HtmlTextWriter(new StringWriter())) {
        writer.Indent = 1;

        writer.AddAttribute("classid", "clsid:D27CDB6E-AE6D-11cf-96B8-444553540000");
        writer.AddAttribute("codebase",
            "http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab#version=6,0,29,0");

        writer.AddAttribute("width", width);
        writer.AddAttribute("height", height);
        writer.RenderBeginTag(HtmlTextWriterTag.Object);

        writer.AddAttribute("name", "movie");
        writer.AddAttribute("value", FlashFile);
        writer.RenderBeginTag(HtmlTextWriterTag.Param);
        writer.RenderEndTag();

        writer.AddAttribute("name", "quality");
        writer.AddAttribute("value", "high");
        writer.RenderBeginTag(HtmlTextWriterTag.Param);
        writer.RenderEndTag();

        writer.AddAttribute("src", FlashFile);
        writer.AddAttribute("quality", "high");
```

```

        writer.AddAttribute("pluginspage", "http://www.macromedia.com/go/getflashplayer");
        writer.AddAttribute("type", "application/x-shockwave-flash");
        writer.AddAttribute("width", width);
        writer.AddAttribute("height", height);
        writer.RenderBeginTag(HtmlTextWriterTag.Embed);
        writer.RenderEndTag();

        writer.RenderEndTag();
        writer.Flush();
        return MvcHtmlString.Create(writer.InnerWriter.ToString());
    }
}

```

Obs.: defina os atributos antes da tag-alvo.

Após importar a namespace, declare o HTML helper no view ASPX:

```

<%@ Page Language="C#" Inherits="System.Web.Mvc.ViewPage<dynamic>" %>
<%@ Import Namespace="AlfredoLotar.Livro.AspnetMvc" %>
<!DOCTYPE html>
<html>
    <head>
        <title>HTML helper Image</title>
    </head>
    <body>
        <div>
            <%: Html.ShowFlash(ResolveUrl("~/Content/arquivo.swf"), "200", "150")%>
        </div>
    </body>
</html>

```

Ou no view Razor:

```

@using AlfredoLotar.Livro.AspnetMvc
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
    <head>
        <title>HTML helper ShowFlash</title>
    </head>
    <body>
        <div>
            @Html.ShowFlash(Href("~/Content/arquivo.swf"), "200", "150")
        </div>
    </body>
</html>

```

Verificações e validações

No ASP.NET MVC, a validação, a verificação e a formatação das informações usadas pela aplicação são realizadas por intermédio de atributos.

Os tipos de atributos de validação e de formatação usados pelo ASP.NET MVC são: `Required`, `StringLength`, `Display`, `DataType`, `DisplayFormat`, `Range`, `RegularExpression`, `Compare`, `Remote`.

O projeto `Data Annotations Extensions` acrescenta 11 novos tipos de atributos ao ASP.NET MVC. Alguns muito úteis, como: `CreditCard`, `Email`, `Url` e `FileExtensions`.

Quando um atributo retorna uma mensagem de erro, usa os métodos `ValidationSummary`, `ValidationMessage` e `ValidationMessageFor` para exibi-la.

Um programador pode usar o método `AddModelError` para exibir mensagens de erro da mesma forma.

Outro recurso interessante dos atributos de validação e formatação é a capacidade de usar arquivos de recursos, ou seja, é possível exibir as mensagens de erro em vários idiomas.

Geralmente, a validação ocorre no cliente e no servidor. Algo essencial para a performance e segurança da aplicação.

8.1 Desenvolvendo uma nova aplicação

Para testar os exemplos deste capítulo, inicie o Visual Studio. Acesse o menu **File** e clique em **New Project ...**.

Na caixa de diálogo **New Project**, selecione **Web > Visual Studio 2012 > ASP.NET MVC 4 Web Application**. Nomeie o projeto como `AppCapítulo8`. Clique em **OK**.

Em seguida, na caixa de combinação **View Engine**, selecione o mecanismo de exibição **Razor**.

Adicione ao projeto o controlador `HomeController`.

No diretório `Models`, crie e adicione entidades para um arquivo `.edmx` a partir do banco de dados `Northwind`. Siga os passos descritos no tópico “4.3 ADO.NET Entity Framework”.

Ao diretório `Models` acrescentamos a classe pública `Employees` com diversas propriedades:

```
using System;
using System.ComponentModel.DataAnnotations;
namespace AlfredoLotar.Livro.AspnetMvc {
    public class Employees {
        [Key]
        public int EmployeeID { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public DateTime BirthDate { get; set; }
        public int Idade { get; set; }
        public string Address { get; set; }
        public string City { get; set; }
        public string Cep { get; set; }
        public string Country { get; set; }
    }
}
```

8.2 Atributos

Os atributos da namespace `System.ComponentModel.DataAnnotations` são usados para formatar, validar e restringir as informações armazenadas nas propriedades.

Exemplo:

```
using System;
using System.ComponentModel.DataAnnotations;
using System.Web.Mvc;

namespace AlfredoLotar.Livro.AspnetMvc {
    public class Employees {
        [Key]
        public int EmployeeID { get; set; }

        [Required(ErrorMessage = "O nome é obrigatório.", AllowEmptyStrings = false)]
        [StringLength(10, MinimumLength = 3)]
        [Display(Name = "Nome")]
        public string FirstName { get; set; }

        [Required(ErrorMessage = "O sobrenome é obrigatório.", AllowEmptyStrings = false)]
        [StringLength(20, MinimumLength = 3)]
        [Display(Name = "Sobrenome")]
        public string LastName { get; set; }

        [Required(ErrorMessage = "A data de nascimento é obrigatória.", AllowEmptyStrings = false)]
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "mm/dd/yy")]
        [Display(Name = "Data de nascimento")]
    }
}
```

```

public DateTime BirthDate { get; set; }

[Required(ErrorMessage = "A idade é obrigatória.", AllowEmptyStrings = false)]
[Range(18, 120)]
public int Idade { get; set; }

[Required(ErrorMessage = "O endereço é obrigatório.", AllowEmptyStrings = false)]
[StringLength(60, MinimumLength = 3)]
[Display(Name = "Endereço")]
public string Address { get; set; }

[Required(ErrorMessage = "O nome da cidade é obrigatório.", AllowEmptyStrings = false)]
[StringLength(15, MinimumLength = 3)]
[Display(Name = "Cidade")]
public string City { get; set; }

[Required(ErrorMessage = "O Cep é obrigatório.", AllowEmptyStrings = false)]
[RegularExpression(@"^\d{5}-\d{3}$", ErrorMessage = "Digite um Cep válido.")]
public string Cep { get; set; }

[Display(Name = "País")]
public string Country { get; set; }

[Required(ErrorMessage = "Informe uma senha.", AllowEmptyStrings = false)]
[DataType(DataType.Password)]
[Display(Name = "Senha")]
public string Password { get; set; }

[Required(ErrorMessage = "Redigite a sua senha.", AllowEmptyStrings = false)]
[Compare("Password", ErrorMessage =
"A nova senha não coincide com a senha de confirmação. Por favor, digite novamente.")]
[DataType(DataType.Password)]
[Display(Name = "Confirme a senha")]
public string ConfirmarPassword { get; set; }
}
}

```

Obs.: quando você não define uma mensagem de erro para um atributo, a mensagem de erro-padrão em inglês é usada.

8.2.1 Atributo Required

O atributo `Required` força a entrada em determinado campo:

```

[Required]
public string FirstName { get; set; }

```

O parâmetro `ErrorMessage` define uma mensagem de erro, enquanto o parâmetro `AllowEmptyStrings` determina se valores em branco são permitidos:

```

[Required(ErrorMessage="O nome é obrigatório.", AllowEmptyStrings = false)]
public string FirstName { get; set; }

```


8.2.2 Atributo StringLength

A quantidade de caracteres é limitada pelo atributo `StringLength`:

```
[StringLength(10, MinimumLength = 3)]  
public string FirstName { get; set; }
```

8.2.3 Atributo Display

O atributo `Display` define o texto visível de uma propriedade quando usada em um campo de formulário, por exemplo:

```
[Display(Name="Nome")]  
public string FirstName { get; set; }
```

Vários atributos podem ser aplicados em uma única propriedade:

```
[Required(ErrorMessage = "O nome é obrigatório.", AllowEmptyStrings = false)]  
[StringLength(10, MinimumLength = 3)]  
[Display(Name = "Nome")]  
public string FirstName { get; set; }
```

Inclusive todos na mesma linha, separados por vírgula:

```
[Required(ErrorMessage = "O nome é obrigatório.", AllowEmptyStrings = false), StringLength(10,  
    MinimumLength = 3), Display(Name = "Nome")]  
public string FirstName { get; set; }
```

8.2.4 Atributo DataType

O atributo `DataType` associa um tipo adicional a uma propriedade. Por exemplo, uma propriedade do tipo `string` pode ter um tipo adicional `Password`, tornando assim o conteúdo da propriedade mais específico:

```
[DataType(DataType.Password)]  
public string Senha { get; set; }
```

Os membros da enumeração `DataType` são: `Custom`, `DateTime`, `Date`, `Time`, `Duration`, `PhoneNumber`, `Currency`, `Text`, `Html`, `MultilineText`, `EmailAddress`, `Password`, `Url`, `ImageUrl`.

8.2.5 Atributo DisplayFormat

Com o atributo `DisplayFormat` é possível aplicar um formato específico a uma data:

```
[DisplayFormat(DataFormatString = "mm/dd/yy")]  
public DateTime BirthDate { get; set; }
```

Formatar um valor monetário:

```
[DisplayFormat(DataFormatString = "{0:C}")]  
public decimal Price { get; set; }
```

Ou definir um texto qualquer para campos em branco:

```
[DisplayFormat(NullDisplayText="[Null]")]  
public decimal Price { get; set; }
```

8.2.6 Atributo Range

O atributo Range aplica um intervalo a uma propriedade. Você precisa definir o valor máximo e mínimo do intervalo:

```
[Range(18, 120)]  
public int Idade { get; set; }
```

8.2.7 Atributo RegularExpression

Para validações mais sofisticadas, mais específicas, use expressões regulares e o atributo RegularExpression.

Exemplo:

```
[RegularExpression(@"^\d{5}-\d{3}$", ErrorMessage = "Digite um Cep válido.")]  
public string Cep { get; set; }
```

Tome cuidado com parênteses em expressões regulares. Leia o artigo “Ataques de negação de serviço de expressão regular e defesas” publicado pela Microsoft.

<http://msdn.microsoft.com/pt-br/magazine/ff646973.aspx>

8.2.8 Atributo Compare

O atributo Compare é novo no ASP.NET MVC e interessante, pois permite comparar o conteúdo de duas propriedades. Por exemplo, a propriedade Password com a propriedade ConfirmarPassword:

```
[Required(ErrorMessage = "Informe uma senha.", AllowEmptyStrings = false)]  
[DataType(DataType.Password)]  
[Display(Name = "Senha")]  
public string Password { get; set; }  
[Required(ErrorMessage = "Redigite a sua senha.", AllowEmptyStrings = false)]  
[DataType(DataType.Password)]  
[Display(Name = "Confirme a senha")]  
[Compare("Password", ErrorMessage =  
    "A nova senha não coincide com a senha de confirmação. Por favor, digite novamente.")]  
public string ConfirmarPassword { get; set; }
```

O atributo Compare pertence à namespace System.Web.Mvc.

8.2.9 Atributo Remote

O atributo `Remote` foi introduzido no ASP.NET MVC 3. Você pode usá-lo, por exemplo, para garantir que as informações de uma propriedade sejam exclusivas. Por exemplo, o Hotmail verifica o prefixo do e-mail que você pretende criar. Do mesmo modo, você pode usar o atributo `Remote` para verificar se determinada informação existe ou não no banco de dados.

O atributo `Remote` usa código criado pelo programador para validar uma propriedade, ou seja, você define o que deve ser validado e como.

A seguir temos uma classe e um método de controlador que verifica se uma categoria já foi cadastrada ou não no banco de dados:

```
using System.Linq;
using System.Web.Mvc;
using AppCapitulo8.Models;
namespace AppCapitulo8.Controllers {
    public class CategoryController : Controller {
        private NorthwindDB db = new NorthwindDB();
        public ActionResult Existe(string CategoryName) {
            try {
                var model = db.Categories.Where(c => c.CategoryName == CategoryName).FirstOrDefault();
                if (model != null)
                    return Json(false, JsonRequestBehavior.AllowGet);
                else
                    return Json(true, JsonRequestBehavior.AllowGet);
            }
            finally {
                if (db != null) db.Dispose();
            }
        }
    }
}
```

Em seguida, aplicamos o atributo `Remote` à propriedade `CategoryName`:

```
[Remote("Existe", "Category", HttpMethod="POST", ErrorMessage = "Categoria existe no BD.")]
public string CategoryName { get; set; }
```

`Existe` é o nome do método de controlador. `Category` é o nome do controlador. Os parâmetros seguintes definem o tipo de operação e a mensagem de erro. Exemplo:

```
using System.ComponentModel.DataAnnotations;
using System.Web.Mvc;
namespace AppCapitulo8.Models {
    [MetadataType(typeof(CategoryMetadata))]
    public partial class Category {}
    public class CategoryMetadata {
        [Required(ErrorMessage = "O nome da categoria é obrigatório.")]
    }
}
```

```

[StringLength(15, MinimumLength = 3,
    ErrorMessage = "Digite no mínimo 3 e no máximo 15 caracteres.")]
[Display(Name = "Categoria")]
[Remote("Existe", "Category", HttpMethod="POST",
    ErrorMessage = "Categoria existe no BD.")]
public string CategoryName { get; set; }

[StringLength(150, MinimumLength = 0,
    ErrorMessage = "Digite no máximo 150 caracteres.")]
[Display(Name = "Descrição")]
public string Description { get; set; }
    }
}

```

Obs.: o parâmetro usado pelo método de controlador Existe deve ser igual ao nome da propriedade ao qual o atributo Remote é aplicado.

8.3 Validação no cliente

Para ativar a validação no cliente é preciso incluir o trecho de código a seguir no arquivo web.config:

```

<appSettings>
    <add key="ClientValidationEnabled" value="true"/>
    <add key="UnobtrusiveJavaScriptEnabled" value="true"/>
</appSettings>

```

E incluir três arquivos JavaScript na master page, layout page ou view:

```

<script src="@Url.Content("~/Scripts/jquery-1.4.4.min.js")" type="text/javascript"></script>
<script src="@Url.Content("~/Scripts/jquery.validate.min.js")" type="text/javascript"></script>
<script src="@Url.Content("~/Scripts/jquery.validate.unobtrusive.min.js")" type="text/javascript">
</script>

```

A chave UnobtrusiveJavaScriptEnabled determina que as regras sejam apresentadas em atributos HTML 5, ou seja, em letras minúsculas, números e hífen e que os atributos usam jQuery para executar a validação.

A validação no cliente pode ser ativada ou desativada também em um método de controlador:

```

HtmlHelper.ClientValidationEnabled = true;
HtmlHelper.UnobtrusiveJavaScriptEnabled = true;

```

Ou em um view ou master page com os métodos EnableClientValidation e EnableUnobtrusiveJavaScript:

```

<%Html.EnableClientValidation(); %>
<%Html.EnableUnobtrusiveJavaScript(); %>

```

```

<% using (Html.BeginForm()) { %>
    <%= Html.ValidationSummary(true) %>
    <fieldset>
    </fieldset>
<% } %>

```

No navegador você pode ver o código-fonte da página e os campos contendo os atributos com as regras de validação no cliente:

```

<div class="editor-field">
<input class="text-box single-line" data-val="true"
    data-val-required="The BirthDate field is required." id="BirthDate" name="BirthDate"
    type="text" value="" />
<span class="field-validation-valid" data-valmsg-for="BirthDate" data-valmsg-replace="true">
</span>
</div>

```

Os atributos com as regras de validação possuem o prefixo `data-val`.

Obs.: a validação no cliente não substitui a validação no servidor. Use sempre `ModelState.IsValid` no método de controlador, pois a validação no cliente pode ser facilmente contornada.

8.4 Exibindo mensagens de erro

O ASP.NET MVC possui alguns HTML helpers que facilitam a exibição de mensagens de erro em um formulário, por exemplo. O método `ValidationSummary` exibe uma lista de campos com erros. Para ativá-lo basta declará-lo no início do formulário:

```

@using (Html.BeginForm()) {
    @Html.ValidationSummary(false, "Ocorreram erros no formulário. Por favor, corrija os erros
    e tente novamente.")
    <fieldset>
        <legend>Funcionários</legend>
        <div class="editor-label">
            @Html.LabelFor(model => model.FirstName)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.FirstName)
        </div>
        <div class="editor-label">
            @Html.LabelFor(model => model.LastName)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.LastName)
        </div>
        <div class="editor-label">

```

```
        @Html.LabelFor(model => model.BirthDate)
    </div>
    <div class="editor-field">
        @Html.EditorFor(model => model.BirthDate)
    </div>

    <div class="editor-label">
        @Html.LabelFor(model => model.Idade)
    </div>
    <div class="editor-field">
        @Html.EditorFor(model => model.Idade)
    </div>

    <div class="editor-label">
        @Html.LabelFor(model => model.Address)
    </div>
    <div class="editor-field">
        @Html.EditorFor(model => model.Address)
    </div>

    <div class="editor-label">
        @Html.LabelFor(model => model.City)
    </div>
    <div class="editor-field">
        @Html.EditorFor(model => model.City)
    </div>

    <div class="editor-label">
        @Html.LabelFor(model => model.Cep)
    </div>
    <div class="editor-field">
        @Html.EditorFor(model => model.Cep)
    </div>

    <div class="editor-label">
        @Html.LabelFor(model => model.Country)
    </div>
    <div class="editor-field">
        @Html.EditorFor(model => model.Country)
    </div>
    <p>
        <input type="submit" value="Create" />
    </p>
</fieldset>
}
```

A saída você observa na figura 8.1:

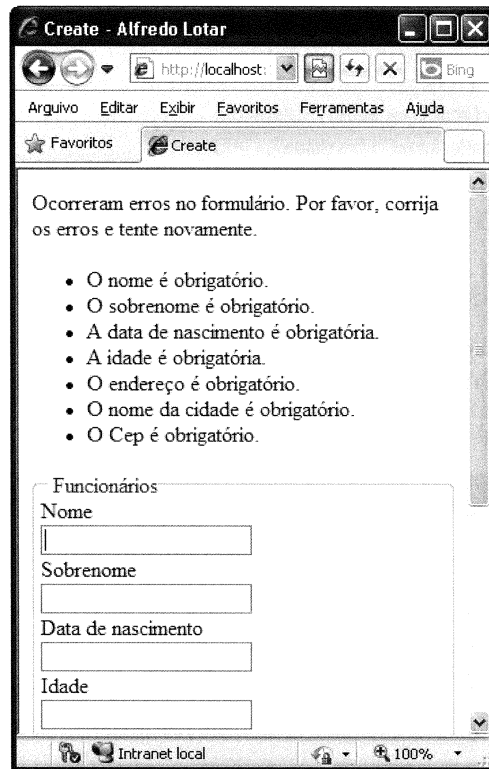


Figura 8.1 – Mensagens de erro exibidas pelo método *ValidationSummary*.

8.4.1 Mensagens de erro com *ValidationMessage*

O método *ValidationSummary* exibe as mensagens de todos os controles no topo do formulário, enquanto o método *ValidationMessage* exibe a mensagem de erro ao lado do controle-alvo:

```
@Html.EditorFor(model => model.FirstName)
@Html.ValidationMessage("FirstName")
```

O segundo parâmetro contém a mensagem de erro ou de orientação:

```
@Html.ValidationMessage("FirstName", "Digite o seu nome.")
```

Quando você exibe todas as mensagens de erro no topo do formulário, marque os campos com erro com um caractere de asterisco. Exemplo:

```
@using (Html.BeginForm()) {
    @Html.ValidationSummary(false,
        "Ocorreram erros no formulário. Por favor, corrija os erros e tente novamente.")
    <fieldset>
        <legend>Employees</legend>
        <div class="editor-label">
```

```

        @Html.LabelFor(model => model.FirstName)
    </div>
    <div class="editor-field">
        @Html.EditorFor(model => model.FirstName)
        @Html.ValidationMessage("FirstName", "*")
    </div>
</fieldset>
}

```

Se preferir, use o método `TextBox` ou `TextBoxFor`:

```

@Html.TextBoxFor(model => model.FirstName)
@Html.ValidationMessage("FirstName", "*")

```

Observe na figura 8.2 a validação de dados em ação com o controle `ValidationMessage`:

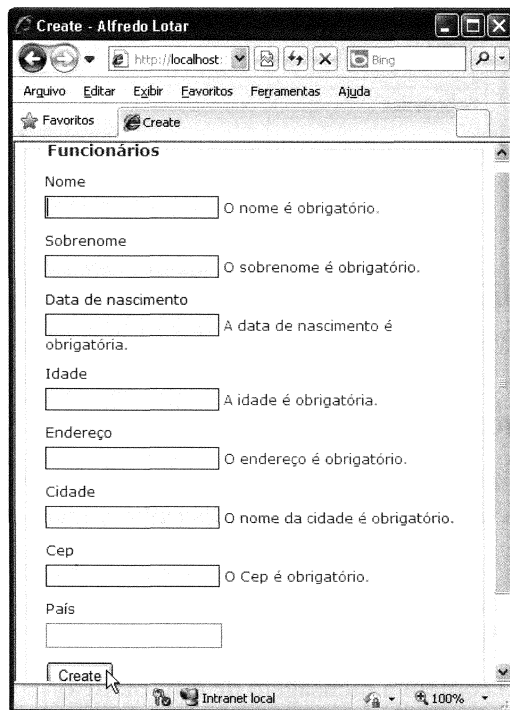


Figura 8.2 – Formulário de validação com propriedades.

8.4.2 Mensagens de erro com `ValidationMessageFor`

O método `ValidationMessageFor` usa expressões lambda. A sua sintaxe é:

```
@Html.ValidationMessageFor(ExpressãoLambada, Mensagem)
```

Exemplo:

```
@Html.ValidationMessageFor(model => model.FirstName, "*")
```


Obs.: a mensagem de erro ou o caractere de asterisco devem ser usados em conjunto com as folhas de estilo abordadas no tópico: “Aplicando folhas de estilos CSS”.

8.4.3 Aplicando folhas de estilo CSS

Quando criamos uma nova aplicação ASP.NET MVC, é acrescentado o arquivo Site.css ao diretório Content com os seguintes seletores CSS de validação:

```
.field-validation-error {
    color: #ff0000;
}

.field-validation-valid {
    display: none;
}

.input-validation-error {
    border: 1px solid #ff0000;
    background-color: #ffebee;
}

.validation-summary-errors {
    font-weight: bold;
    color: #ff0000;
}

.validation-summary-valid {
    display: none;
}
```

Para aplicá-los aos campos do formulário, basta inserir a linha a seguir ao view:

```
<link href="@Url.Content("~/Content/Site.css")" rel="stylesheet" type="text/css"/>
```

Exemplo:

```
<html>
  <head>
    <title>@ViewBag.Title</title>
    <link href="@Url.Content("~/Content/Site.css")" rel="stylesheet" type="text/css"/>
  </head>
  <!--Restante do código HTML-->
```

8.5 Data Annotations Extensions

O projeto Data Annotations Extensions acrescenta 11 atributos de validação adicionais, conforme lista a seguir:

- CreditCard
- Date

- Digits
- Email
- EqualTo
- FileExtensions
- Integer
- Max
- Min
- Numeric
- Url

Obtenha mais informações sobre o projeto Data Annotations Extensions no website:

<http://dataannotationsextensions.org/>

É necessário instalar o Data Annotations Extensions a partir do Visual Studio 2010. Clique com o botão direito do mouse em **References** e selecione **Manage NuGet Packages....** Observe a figura 8.3.

Conecte o computador à Internet, em seguida, na caixa de diálogo **Manage NuGet Packages...**, procure por DataAnnotationsExtensions e instale o projeto DataAnnotationsExtensions.MVC3. Observe a figura 8.4.

Obs.: antes de instalar o Data Annotations Extensions verifique se o seu computador possui o Windows Power Shell 2.0. Caso contrário, instale-o a partir do website da Microsoft.

Após a instalação do projeto DataAnnotationsExtensions.MVC3, aplique os atributos às propriedades, conforme o exemplo a seguir:

```
public class Clientes {
    [Email(ErrorMessage = "Digite um e-mail válido.")]
    [Required(ErrorMessage = "E-mail é obrigatório.")]
    public string Email { get; set; }

    [CreditCard(ErrorMessage="Digite um número de cartão de crédito válido.")]
    public string CartaoCredito { get; set; }

    [Url(ErrorMessage = "Digite uma URL válida.")]
    public string Website { get; set; }

    [FileExtensions("png|jpg|jpeg|gif", ErrorMessage = "Extensões permitidas:
    png|jpg|jpeg|gif")]
    public string Foto { get; set; }
}
```

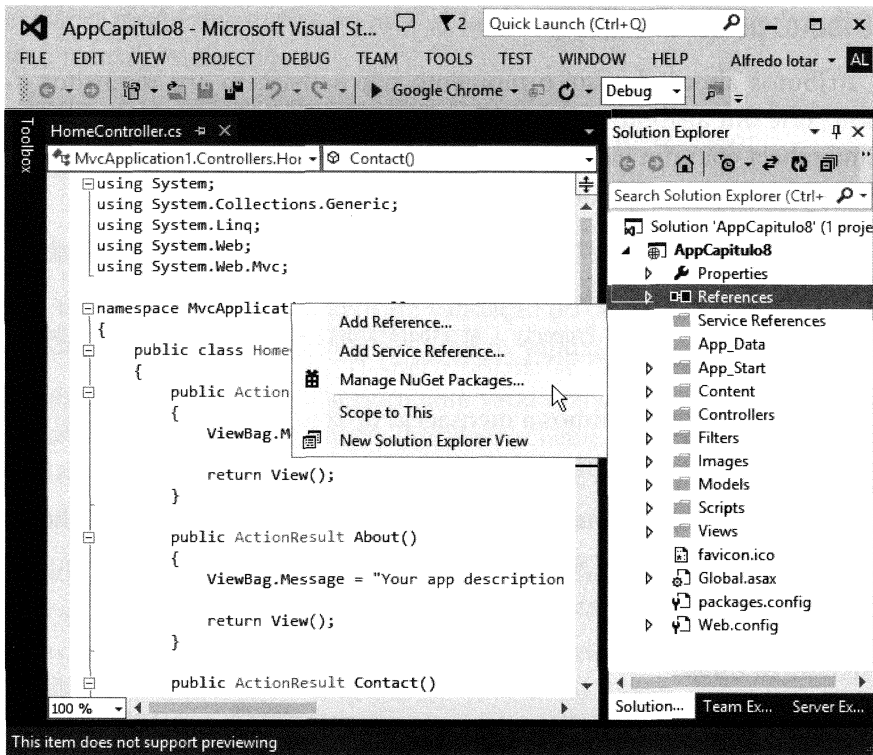


Figura 8.3 – Passos para adicionar o Data Annotations Extensions.

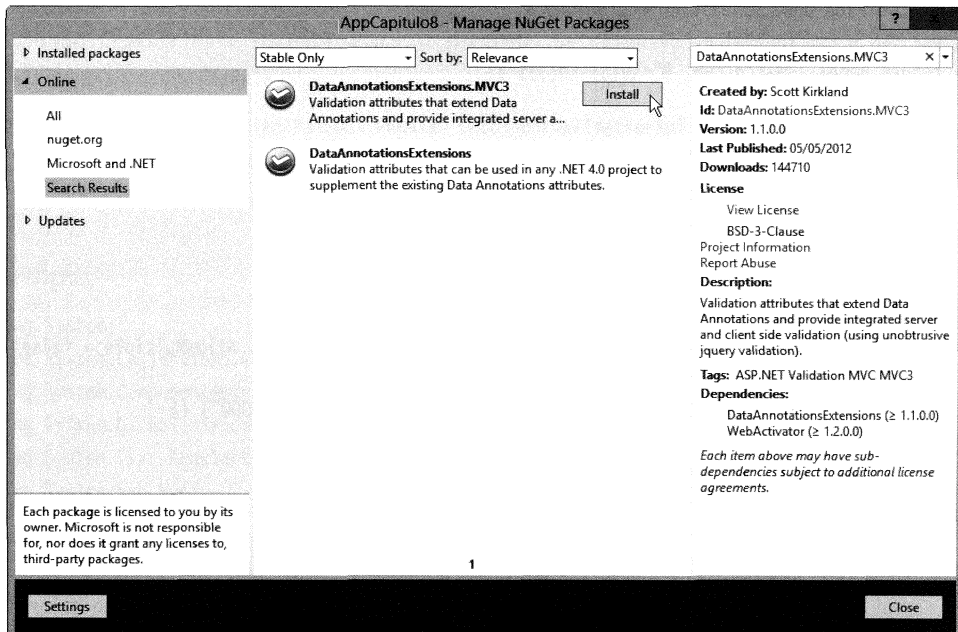


Figura 8.4 - Caixa de diálogo Add Library Package Reference.

8.6 Atributos personalizados

Se os atributos abordados até o momento não satisfazem aos requisitos da sua aplicação, crie um atributo personalizado. Por exemplo, uma aplicação geralmente tem uma tabela com um campo denominado Cep. Podemos validar e verificar uma propriedade Cep com um atributo personalizado.

O atributo personalizado é derivado da classe `ValidationAttribute` e aplicado a propriedades e campos:

```
[AttributeUsage(AttributeTargets.Property | AttributeTargets.Field, AllowMultiple = false)]
sealed public class CepAttribute : ValidationAttribute { }
```

No método construtor definimos a mensagem de erro padrão:

```
public CepAttribute() : base("Digite um cep válido. Ex.: 96100-000") { }
```

Em seguida, modificamos o método `IsValid` e adicionamos a lógica de validação. O método `IsValid` retorna `true` se a validação for bem-sucedida.

```
public override bool IsValid(object value) {
    bool resultado = true;
    // Restante do código;
    return resultado;
}
```

Exemplo:

```
public override bool IsValid(object value) {
    var cep = (string)value;
    return Regex.IsMatch(cep, @"^\d{5}-\d{3}$");
}
```

Observe a seguir o código completo do atributo `CepAttribute`:

```
using System;
using System.ComponentModel.DataAnnotations;
using System.Text.RegularExpressions;

namespace AppCapitulo8.Models {
    [AttributeUsage(AttributeTargets.Property | AttributeTargets.Field, AllowMultiple = false)]
    sealed public class CepAttribute : ValidationAttribute {
        public CepAttribute() : base("Digite um cep válido. Ex.: 96100-000") { }
        public override bool IsValid(object value) {
            var cep = (string)value;
            return Regex.IsMatch(cep, @"^\d{5}-\d{3}$");
        }
    }
}
```

Se necessário, modifique outros membros da classe `ValidationAttribute`. Por exemplo, formate a mensagem de erro:

```
public override string FormatErrorMessage(string name) {
    return String.Format(CultureInfo.CurrentCulture, ErrorMessageString, name);
}
```

8.6.1 Validação do lado do cliente

Os atributos do ASP.NET MVC aplicam validação do lado do cliente e do servidor. Quando criamos os nossos próprios atributos de validação, podemos fazer o mesmo.

O atributo `CepAttribute` criado anteriormente valida os dados somente no servidor. Para que a validação aconteça também do lado do cliente, implementamos a interface `IClientValidatable`:

```
sealed public class CepAttribute : ValidationAttribute, IClientValidatable {....}
```

Em seguida, criamos a classe `ModelClientValidationCepRule` que herda as funções de `ModelClientValidationRule`:

```
public class ModelClientValidationCepRule : ModelClientValidationRule {
    public ModelClientValidationCepRule(string errorMessage) {
        base.ErrorMessage = errorMessage;
        base.ValidationType = "cep";
    }
}
```

A classe `CepAttribute` implementa o método `GetClientValidationRules` e a classe `ModelClientValidationCepRule`:

```
public IEnumerable<ModelClientValidationRule> GetClientValidationRules(ModelMetadata metadata,
    ControllerContext context) {
    yield return new ModelClientValidationCepRule(FormatErrorMessage(metadata.GetDisplayName()));
}
```

Exemplo:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.Globalization;
using System.Text.RegularExpressions;
using System.Web.Mvc;

namespace AppCapitulo8.Models {
    [AttributeUsage(AttributeTargets.Property | AttributeTargets.Field, AllowMultiple = false)]
    sealed public class CepAttribute : ValidationAttribute, IClientValidatable {
        public CepAttribute() : base("Digite um cep válido. Ex.: 96100-000") { }
        public override bool IsValid(object value) {
```

```

        var cep = (string)value;
        return Regex.IsMatch(cep, @"^\d{5}-\d{3}$");
    }
    public IEnumerable<ModelClientValidationRule>
    GetClientValidationRules(ModelMetadata metadata, ControllerContext context) {
        yield return new ModelClientValidationCepRule(FormatErrorMessage(metadata.GetDisplayName()));
    }
}
}

```

No arquivo Scripts\ValidationCep.js temos a lógica de validação do lado do cliente:

```

(function ($) {
    $.validator.addMethod(
        "Cep",
        function (value, element) {
            var cepPattern = /^d{5}-d{3}$/;
            return (!value && this.optional(element)) || cepPattern.test(value);
        },
        "Digite um cep válido. Ex.: 96100-000");
    $.validator.unobtrusive.adapters.addBool("Cep");
} (jQuery));

```

Obs.: Cep é o prefixo do atributo.

8.6.2 Aplicando o atributo personalizado

A implementação do atributo personalizado é simples. Exemplo:

```

public class Clientes {
    [Cep(ErrorMessage = "Digite um cep válido.")]
    public string Cep { get; set; }
}

```

No view é declarado o arquivo ValidationCep.js que contém as regras de validação do lado do cliente:

```
<script src="@Url.Content("~/Scripts/ValidationCep.js")" type="text/javascript"></script>
```

Exemplo:

```

<!DOCTYPE html>
<html>
    <head>
        <title>Create</title>
        <script src="@Url.Content("~/Scripts/jquery-1.4.4.min.js")" type="text/javascript">
        </script>
        <script src="@Url.Content("~/Scripts/jquery.validate.min.js")" type="text/javascript">
        </script>

```

```

<script src="@Url.Content("~/Scripts/jquery.validate.unobtrusive.min.js")"
    type="text/javascript"></script>
<script src="@Url.Content("~/Scripts/ValidationCep.js")" type="text/javascript"></script>
<link href="@Url.Content("~/Content/Site.css")" rel="stylesheet" type="text/css"/>
</head>
<body>
    .....
</body>
</html>

```

8.7 Validando propriedades do ADO.NET Entity Framework

Para validar as propriedades de uma classe do ADO.NET Entity Framework crie uma nova classe parcial com o mesmo nome e sob a mesma namespace (importantíssimo isso):

```

using System.ComponentModel.DataAnnotations;
namespace AppCapítulo8.Models {
    public partial class Category {}
}

```

Defina o atributo `MetadataType`:

```

[MetadataType(typeof(CategoryMetadata))]
public partial class Category {}

```

A classe `CategoryMetadata` contém as propriedades com os atributos de validação e formatação:

```

public class CategoryMetadata {
    [Display(Name = "Categoria")]
    public string CategoryName { get; set; }
}

```

Exemplo:

```

using System.ComponentModel.DataAnnotations;
namespace AppCapítulo8.Models {
    [MetadataType(typeof(CategoryMetadata))]
    public partial class Category {}

    public class CategoryMetadata {
        [Required(ErrorMessage = "O nome da categoria é obrigatório.")]
        [StringLength(15, MinimumLength = 3,
            ErrorMessage = "Digite no mínimo 3 e no máximo 15 caracteres.")]
        [Display(Name = "Categoria")]
        public string CategoryName { get; set; }
    }
}

```

```

[StringLength(150, MinimumLength = 0,
    ErrorMessage = "Digite no máximo 150 caracteres.")]
[Display(Name = "Descrição")]
public string Description { get; set; }
}
}

```

Obs.: repita o processo para as demais classes, ou seja, crie uma classe *partial* e outra com o sufixo *Metadata* para cada classe do ADO.NET Entity Framework.

8.8 Validação com ModelState

O método `AddModelError` também exibe mensagens de erro. Observe o código a seguir:

```

public ActionResult Create() {
    return View();
}

[HttpPost]
public ActionResult Create(Employees entidade) {
    if (string.IsNullOrEmpty(entidade.FirstName) || !Regex.IsMatch(entidade.FirstName,
        @"^[a-zA-Z\s]{3,10}$")) {
        ModelState.AddModelError("FirstName", "O nome é obrigatório.");
    }
    if (string.IsNullOrEmpty(entidade.LastName) || !Regex.IsMatch(entidade.LastName,
        @"^[a-zA-Z\s]{3,20}$")) {
        ModelState.AddModelError("LastName", "O sobrenome é obrigatório.");
    }

    if (!ModelState.IsValid) {
        ModelState.AddModelError("",
            "Ocorreram erros no formulário. Por favor, corrija os erros e tente novamente.");
    }
    return View();
}

```

Você pode exibir a mensagem de erro ao lado do campo do formulário:

```
ModelState.AddModelError("FirstName", "O nome é obrigatório.");
```

E uma mensagem genérica no topo do formulário:

```

ModelState.AddModelError("",
    "Ocorreram erros no formulário. Por favor, corrija os erros e tente novamente.");

```

Observe a figura 8.5:

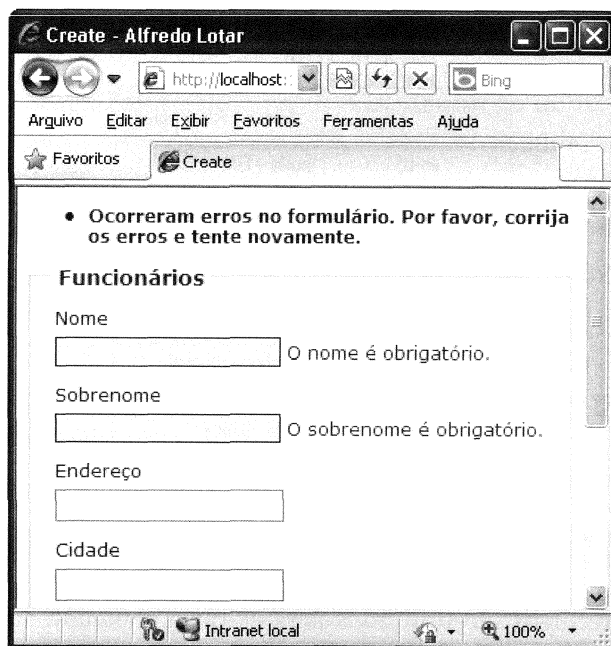


Figura 8.5 – Validação com ModelState.

Obs.: se você define regras de validação em uma propriedade e também com `ModelState`, as mensagens de erro definidas na propriedade são exibidas no formulário.

8.9 Internacionalização de mensagens de erro

Quando você não define uma mensagem de erro para um atributo de validação, a versão em inglês é usada. Para exibir a mensagem em português, defina-a explicitamente.

Se a aplicação necessita exibir mensagem de erro em vários idiomas, use arquivos de recursos. Um para cada idioma.

O arquivo de recursos carregado pela aplicação é determinado pelo nome da cultura formado pelo padrão:

<CódigoDoIdioma>-<País/CódigoDaRegião>

De modo que o `CódigoDoIdioma` é composto de duas letras minúsculas, e o `País/CódigoDaRegião` é composto de duas letras maiúsculas.

Exemplo:

en-US

pt-BR

es-UY

A seguir, listamos os nomes de algumas culturas predefinidas:

Nome da cultura (culture name)	Idioma – País/Região
en-US	Inglês – EUA
en-CA	Inglês – Canadá
en-AU	Inglês – Austrália
pt-BR	Português – Brasil
pt-PT	Português – Portugal
es-AR	Espanhol – Argentina
es-UY	Espanhol – Uruguai
de-DE	Alemão – Alemanha
zh-CN	Chinês – China
ja-JP	Japonês – Japão

8.9.1 Resource files – arquivos de recursos

Os arquivos de recursos são utilizados para exibir o conteúdo apropriado para o usuário no seu próprio idioma. Um website pode ser formatado para exibir conteúdo em múltiplos idiomas. Assim, você cria um arquivo de recursos que contém as legendas dos controles, as mensagens de erro, as imagens e o conteúdo em geral no idioma inglês, bem como outro arquivo de recursos que contém o mesmo conteúdo em português.

O conteúdo pode ser exibido com base no idioma configurado no computador do usuário ou de forma explícita pelo próprio usuário.

Um arquivo de recursos (.resx) é um arquivo no formato XML que contém strings que podem ser traduzidas em diferentes idiomas. Você pode criar um arquivo de recursos para cada idioma suportado pelo website. Em tempo de execução, esse arquivo .resx é compilado em um assembly. Assemblies que têm somente arquivos de recursos são chamados de satellite assemblies.

Podemos criar um arquivo de recursos global, o qual pode ser lido a partir de qualquer página. Esse arquivo deve ser colocado no diretório reservado `App_GlobalResources` e nomeado da seguinte forma:

```
nome.resx  
nome.idioma.resx  
nome.idioma-cultura.resx
```

Exemplo:

```
Default.resx  
Default.pt.resx  
Default.pt-br.resx
```

Para criar e usar um arquivo de recursos, siga os seguintes passos:

- Abra o Visual Studio e crie um novo projeto.
- Na janela **Solution Explorer**, clique com o botão direito do mouse no nome do projeto e selecione a opção **Add ASP.NET Folder**; em seguida, selecione a opção **App_GlobalResources**.
- Clique com o botão direito do mouse em **App_GlobalResources** e acesse **New Item....**
- Na caixa de diálogo **Add New Item**, selecione **Resources File**. Utilize o nome-padrão **Resource1.resx**. Repita os passos anteriores e adicione também o arquivo **Resource1.en-us.resx**.
- Dê um duplo clique no arquivo de recursos criado. Em seguida, preencha os campos **Name** e **Value**.

Observe a figura 8.6:

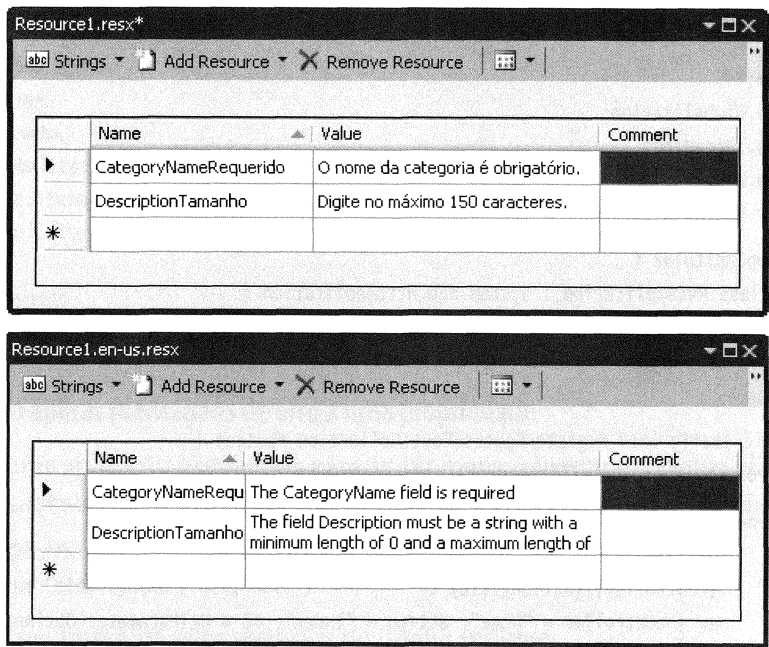


Figura 8.6 – Arquivos de recursos com conteúdo em português e inglês.

8.9.2 Determine a cultura

Para que o ASP.NET MVC possa exibir o arquivo de recursos correto, é necessário que você utilize um mecanismo para identificar o idioma utilizado pelo usuário.

Esse mecanismo pode ser manual – o usuário seleciona o idioma do website clicando em um link. Ou o ASP.NET MVC pode detectar o idioma usado pelo computador do usuário por intermédio de um filtro de ação ou do arquivo `Global.asax` definindo o método `Application_BeginRequest`, o qual é invocado pelo evento `BeginRequest`:

```
protected void Application_BeginRequest(Object sender, EventArgs e) {
    if (Request.UserLanguages != null) {
        Thread.CurrentThread.CurrentCulture =
            CultureInfo.CreateSpecificCulture(Request.UserLanguages[0]);
        Thread.CurrentThread.CurrentUICulture = Thread.CurrentThread.CurrentCulture;
    }
    else {
        Thread.CurrentThread.CurrentCulture = new CultureInfo("pt-BR");
        Thread.CurrentThread.CurrentUICulture = Thread.CurrentThread.CurrentCulture;
    }
}
```

Exemplo:

```
using System;
using System.Globalization;
using System.Threading;
using System.Web.Mvc;
using System.Web.Routing;

namespace AppCapitulo8 {
    public class MvcApplication : System.Web.HttpApplication {
        public static void RegisterGlobalFilters(GlobalFilterCollection filters) {
            filters.Add(new HandleErrorAttribute());
        }

        public static void RegisterRoutes(RouteCollection routes) {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

            routes.MapRoute(
                "Default",
                "{controller}/{action}/{id}",
                new { controller = "Home", action = "Index", id = UrlParameter.Optional }
            );
        }

        protected void Application_Start() {
            AreaRegistration.RegisterAllAreas();
            RegisterGlobalFilters(GlobalFilters.Filters);
            RegisterRoutes(RouteTable.Routes);
        }
    }
}
```

```
protected void Application_BeginRequest(Object sender, EventArgs e) {
    if (Request.UserLanguages != null) {
        Thread.CurrentThread.CurrentCulture =
            CultureInfo.CreateSpecificCulture(Request.UserLanguages[0]);
        Thread.CurrentThread.CurrentUICulture = Thread.CurrentThread.CurrentCulture;
    }
    else {
        Thread.CurrentThread.CurrentCulture = new CultureInfo("pt-BR");
        Thread.CurrentThread.CurrentUICulture = Thread.CurrentThread.CurrentCulture;
    }
}
}
```

A propriedade `CultureInfo` é definida com uma cultura predefinida, enquanto a propriedade `CurrentUICulture` determina qual arquivo de recursos será carregado pela página.

A cultura pode ser definida também no arquivo de configuração `web.config`:

```
<?xml version="1.0"?>
<configuration>
  <system.web>
    <globalization
      culture="pt-BR"
      uiCulture="pt-BR"
    />
  </system.web>
</configuration>
```

8.9.3 Como aplicar mensagens de erro a uma propriedade

Quando uma aplicação é projetada para exibir mensagens em um único idioma declaramos as mensagens de erro nas propriedades de forma explícita no idioma desejado:

```
[Required(ErrorMessage = "O nome da categoria é obrigatório.")]
public string CategoryName { get; set; }
```

Ou de forma implícita para usuários de língua inglesa:

```
[Required]
public string CategoryName { get; set; }
```

Aplicações que leem mensagens de erro a partir de arquivos de recursos definem as propriedades `ErrorMessageResourceName` e `ErrorMessageResourceType`. `ErrorMessageResourceName` contém a chave relacionada à mensagem de erro. E `ErrorMessageResourceType` determina o nome do arquivo de recursos carregado.

Exemplo:

```
[Required(ErrorMessageResourceType = typeof(Resource1),  
    ErrorMessageResourceName = "CategoryNameRequerido")]  
public string CategoryName { get; set; }
```

Para exibir o conteúdo de uma chave em um view, use:

```
@Resources.Resource1.CategoryNameRequerido
```

A mesma sintaxe se aplica quando elaboramos links:

```
@Html.ActionLink(Resources.Resource1.Button, "Create")
```

8.9.4 Testando a aplicação

Para testar a aplicação, acesse o painel de controle do Windows e altere o idioma e país na caixa de diálogo Opções regionais e de idioma. Mude para Inglês (Estados Unidos). Observe a figura 8.7:

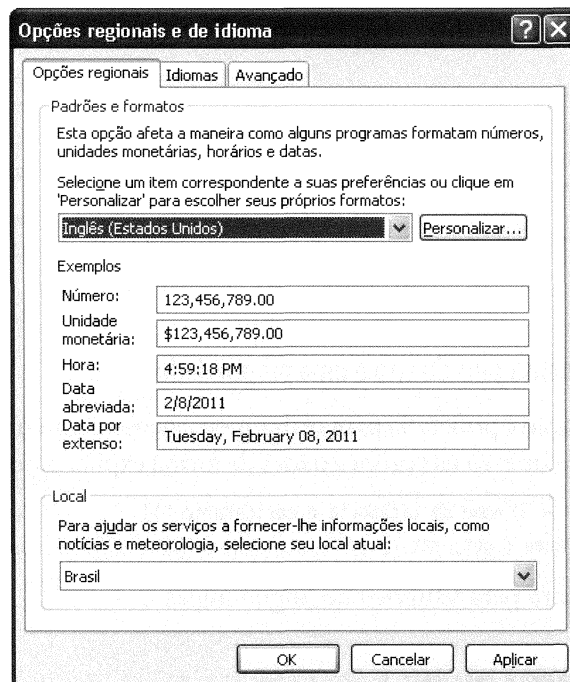


Figura 8.7 - Caixa de diálogo opções regionais e de idioma.

Rode a aplicação e veja a diferença entre as mensagens de erro. Observe as figuras 8.8 e 8.9:

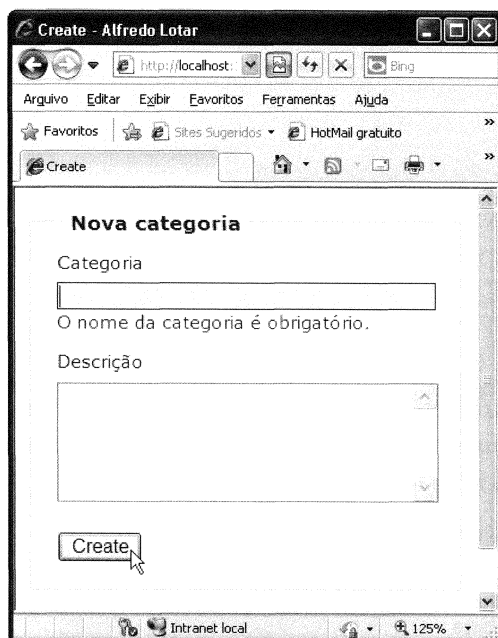


Figura 8.8 – Mensagens de erro em português.

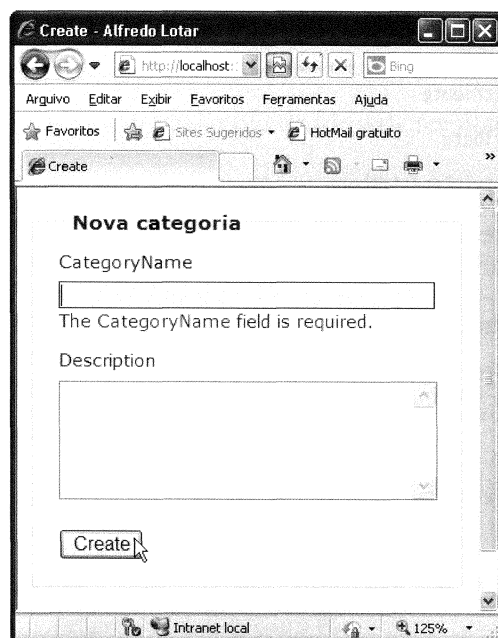


Figura 8.9 – Mensagens de erro em inglês.

Filtros de ação

Filtro de ação é um atributo anexado a uma classe ou método de controlador injetando código extra antes ou após o método de controlador ser executado, ou antes, e após o resultado da ação ser retornado.

O ASP.NET MVC possui oito filtros de ação predefinidos:

- `AuthorizeAttribute`
- `RequireHttpsAttribute`
- `OutputCacheAttribute`
- `HandleErrorAttribute`
- `AsyncTimeoutAttribute`
- `ChildActionOnlyAttribute`
- `ValidateInputAttribute`
- `ValidateAntiForgeryTokenAttribute`

Os filtros de ação predefinidos e os filtros de ação personalizados implementam várias interfaces.

A seguir listamos as interfaces usadas pelos filtros de ação:

- `IAuthorizationFilter`
- `IActionFilter`
- `IResultFilter`
- `IExceptionFilter`

9.1 Filtros de autorização

Os filtros de autorização tomam decisões de segurança sobre a execução de um método de controlador e são derivados da classe `FilterAttribute` e da interface `IAuthorizationFilter`.

Os filtros de ação derivados da classe `FilterAttribute` são executados antes do método de controlador ser executado.

Exemplos de filtros de autorização do ASP.NET MVC:

- `AuthorizeAttribute`
- `RequireHttpsAttribute`
- `ValidateInputAttribute`
- `ValidateAntiForgeryTokenAttribute`
- `ChildActionOnlyAttribute`

9.2 Filtros de ação

Implementam `IActionFilter`. A interface `IActionFilter` possui dois métodos: `OnActionExecuting` e `OnActionExecuted`.

`OnActionExecuting` é executado antes do método de controlador. `OnActionExecuted` é executado após o método de controlador.

9.3 Filtros de resultado

Implementam `IResultFilter`. A interface `IResultFilter` possui dois métodos: `OnResultExecuting` e `OnResultExecuted`.

`OnResultExecuting` é executado antes do resultado da ação. `OnResultExecuted` é executado após o resultado da ação.

O atributo `OutputCacheAttribute` é um exemplo de filtro de resultado.

9.4 Filtros de exceções

Implementam a interface `IExceptionHandler` e são executados quando uma exceção ocorre.

O atributo `HandleErrorAttribute` é um exemplo de filtro de exceção, pois é derivado de `FilterAttribute` e implementa `IExceptionHandler`.

9.5 Desenvolvendo uma nova aplicação

Para testar os exemplos deste capítulo, inicie o Visual Studios. Acesse o menu **File** e clique em **New Project**

Na caixa de diálogo **New Project**, selecione **Web > Visual Studio 2012 > ASP.NET MVC 4 Web Application**. Nomeie o projeto como `AppCapitulo9`. Clique em **OK**.

Em seguida, na caixa de combinação **View Engine**, selecione o mecanismo de exibição **Razor**.

Adicione ao projeto o controlador `HomeController`.

9.6 Criando um filtro de ação personalizado

Geralmente, um filtro de ação personalizado é derivado de `ActionFilterAttribute`, mas pode ser derivado também de `FilterAttribute` como `AuthorizeAttribute`.

A diferença está no momento de execução. Os filtros de ação derivados de `FilterAttribute` são executados sempre antes do método de controlador. Útil quando verificações de segurança são necessárias.

Os filtros de ação personalizados derivados de `ActionFilterAttribute` são executados:

- Imediatamente antes da execução do método de controlador.
- Imediatamente após a execução do método de controlador.
- Imediatamente antes do resultado do método de controlador.
- Imediatamente após o resultado do método de controlador.

Observe a seguir a declaração da classe `ActionFilterAttribute`:

```
[AttributeUsageAttribute(AttributeTargets.Class|AttributeTargets.Method, Inherited = true,
    AllowMultiple = false)]
public abstract class ActionFilterAttribute : FilterAttribute, IActionFilter, IResultFilter {
    public virtual void OnActionExecuted(ActionExecutedContext filterContext);
    public virtual void OnActionExecuting(ActionExecutingContext filterContext);
    public virtual void OnResultExecuted(ResultExecutedContext filterContext);
    public virtual void OnResultExecuting(ResultExecutingContext filterContext);
}
```

Para adicionar um novo filtro de ação personalizado ao projeto, adicione ao diretório `Models` uma nova classe. Exemplo: `MessageFilterAttribute.cs`

Ao clicar em **Add** o código a seguir é adicionado ao arquivo `MessageFilterAttribute.cs`:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

namespace AppCapitulo9.Models {
    public class MessageFilterAttribute {
    }
}
```

Altere o código gerado. A classe `MessageFilterAttribute` é derivada de `ActionFilterAttribute`:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
```

```
namespace AppCapitulo9.Models {
    public class MessageFilterAttribute: ActionFilterAttribute {
    }
}
```

Em seguida, substitua os métodos `OnActionExecuted`, `OnActionExecuting`, `OnResultExecuted` e `OnResultExecuting`. Você não precisa escrever código para todos os métodos. Exemplo:

```
public string Mensagem { get; set; }
public override void OnActionExecuting(ActionExecutingContext filterContext) {
    filterContext.HttpContext.Response.Write(Mensagem);
}
```

A linha:

```
filterContext.HttpContext.Response.Write(Mensagem);
```

Pode ser substituída por:

```
filterContext.Result = new ContentResult {
    Content = Mensagem,
    ContentEncoding = System.Text.Encoding.UTF8
};
```

Exemplo:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
namespace AppCapitulo9.Models {
    public class MessageFilterAttribute: ActionFilterAttribute {
        public string Mensagem { get; set; }
        public override void OnActionExecuting(ActionExecutingContext filterContext) {
            filterContext.Result = new ContentResult {
                Content = Mensagem,
                ContentEncoding = System.Text.Encoding.UTF8
            };
        }
    }
}
```

Caso o método retorne um tipo `bool` invoque o método da classe-base.

```
base.OnActionExecuting(filterContext);
```

Exemplo:

```
namespace AppCapitulo9.Models {
    public class MessageFilterAttribute: ActionFilterAttribute {
```

```

        public string Mensagem { get; set; }
        public override void OnActionExecuting(ActionExecutingContext filterContext) {
            if (filterContext.HttpContext.Request.IsSecureConnection) {
                filterContext.Result = new ContentResult {
                    Content = Mensagem,
                    ContentEncoding = System.Text.Encoding.UTF8
                };
            }
            base.OnActionExecuting(filterContext);
        }
    }
}

```

Após compilar a aplicação, aplicar o filtro de ação ao método de controlador é fácil. Declare a namespace no topo:

```

using AppCapitulo9.Models;
namespace AppCapitulo9.Controllers {
    public class HomeController : Controller {}
}

```

Em seguida, aplique o filtro de ação e defina os parâmetros, se necessário:

```

public class HomeController : Controller {
    [MessageFilter(Mensagem="Olá mundo!")]
    public ActionResult Index(string a) {
        return View();
    }
}

```

Se você pretende desenvolver um filtro que restringe o acesso dos usuários a um determinado método de controlador, crie um filtro de ação que herda as funções da classe `AuthorizeAttribute`.

Exemplo:

```

using System;
using System.Web.Mvc;
namespace AppCapitulo9.Models {
    [AttributeUsage(AttributeTargets.Class | AttributeTargets.Method, Inherited = true,
        AllowMultiple = false)]
    public class SecurityAttribute : AuthorizeAttribute {
        protected override bool AuthorizeCore(System.Web.HttpContextBase httpContext) {
            if (httpContext.Request.IsAuthenticated && httpContext.Request.IsSecureConnection)
                return true;
            return base.AuthorizeCore(httpContext);
        }
    }
}

```

A ordem em que o filtro de ação é aplicado pode ser definida por intermédio do parâmetro `Order`.

Exemplo:

```
[HandleError(Order=1)]
[OutputCache(Order=2)]
public ActionResult Index() {
    return View();
}
```

Se você pretende aplicar um filtro de ação personalizado a todos os controladores da aplicação, use um recurso do ASP.NET MVC 3 denominado Global Filters. Para aplicar um Global Filters registre no arquivo `Global.asax` a classe que contém o código do filtro de ação:

```
filters.Add(new MessageFilterAttribute());
```

Exemplo:

```
using System.Web.Mvc;
using System.Web.Routing;
using AppCapitulo9.Models;

namespace AppCapitulo9 {
    public class MvcApplication : System.Web.HttpApplication {
        public static void RegisterGlobalFilters(GlobalFilterCollection filters) {
            filters.Add(new HandleErrorAttribute());
            filters.Add(new MessageFilterAttribute());
        }

        public static void RegisterRoutes(RouteCollection routes) {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
            routes.MapRoute(
                "Default",
                "{controller}/{action}/{id}",
                new { controller = "Home", action = "Index", id = UrlParameter.Optional }
            );
        }

        protected void Application_Start() {
            AreaRegistration.RegisterAllAreas();
            RegisterGlobalFilters(GlobalFilters.Filters);
            RegisterRoutes(RouteTable.Routes);
        }
    }
}
```

Recurso útil quando temos um filtro de ação que compacta as páginas de um website ou grava informações no log.

O modo de programação Ajax proporciona uma experiência agradável ao usuário, pois somente partes da página são atualizadas.

Por exemplo, você clica em um item de menu e no centro da página é carregado o conteúdo, ou seja somente o conteúdo é carregado. O logo, os anúncios, o menu permanecem estáticos. Não há o famoso refresh da página. É semelhante a um software de bate-papo em que o conteúdo aparece na tela, sem necessidade de recarregar a página.

Implementar o modo de programação Ajax no ASP.NET MVC é muito fácil, como veremos no decorrer deste capítulo.

10.1 Desenvolvendo uma nova aplicação

Para testar os exemplos deste capítulo, inicie o Visual Studio. Acesse o menu **File** e clique em **New Project ...**.

Na caixa de diálogo **New Project**, selecione **Web > Visual Studio 2012 > ASP.NET MVC 4 Web Application**. Nomeie o projeto como **AppCapítulo10**. Clique em **OK**.

Em seguida, na caixa de combinação **View Engine**, selecione o mecanismo de exibição **Razor**. Adicione ao projeto o controlador **HomeController**.

No diretório **Models**, crie e adicione entidades para um arquivo **.edmx** a partir do banco de dados **Northwind**. Siga os passos descritos no tópico “4.3 ADO.NET Entity Framework”.

10.2 Ativando Unobtrusive Ajax

O ASP.NET MVC usa o chamado **unobtrusive Ajax**. O que significa que as tags HTML geradas são compatíveis com a HTML 5, e os atributos têm o prefixo **data-** e usam **jQuery**. Observe a seguir:

```
<a data-ajax="true" data-ajax-mode="replace" data-ajax-update="#divProdutos"
  href="/Home/ListaProdutos/Beverages">Beverages</a>
```

Unobtrusive Ajax é ativado no arquivo `web.config`:

```
<configuration>
  <appSettings>
    <add key="UnobtrusiveJavaScriptEnabled" value="true"/>
  </appSettings>
</configuration>
```

E no view ou master page:

```
<script src="@Url.Content("~/Scripts/jquery-1.4.4.js")" type="text/javascript"></script>
<script src="@Url.Content("~/Scripts/jquery.unobtrusive-ajax.js")" type="text/javascript">
</script>
```

Obs.: para rodar aplicações Ajax, o navegador do usuário deve oferecer suporte para JavaScript.

10.3 Método Ajax.ActionLink

O método `Ajax.ActionLink` é um Ajax helper, ou seja, invoca um método de controlador e retorna o conteúdo de forma assíncrona:

A principal sintaxe do método `Ajax.ActionLink` é:

```
@Ajax.ActionLink(TextoLink, MétododeControlador, parâmetros, AjaxOptions)
```

Exemplo:

```
<ul style="display: inline">
  @foreach (var item in Model) {
    <li style="display: inline">
      @Ajax.ActionLink(item.CategoryName, "ListaProdutos", new {
        categoria = item.CategoryName }, new AjaxOptions { UpdateTargetId = "divProdutos" })
    </li>
  }
</ul>
```

A tag `div` com `id` igual a `divProdutos` é definida na propriedade `UpdateTargetId` e é usada para carregar as informações no navegador:

```
<div id="divProdutos">
  -----
</div>
```

A figura 10.1 mostra um fluxograma contendo os passos necessários para a elaboração de um exemplo que exibe uma lista de categorias com links e seus respectivos produtos.

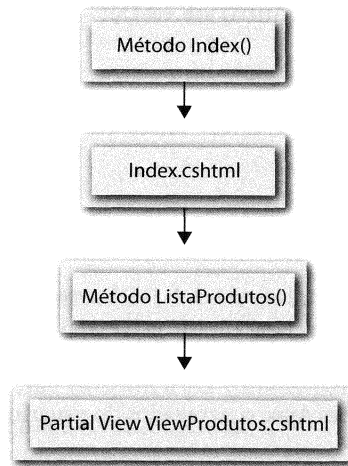


Figura 10.1 – Fluxograma lista de produtos por categoria.

O método de controlador Index carrega a lista de categorias usadas nos links:

```

private NorthwindBD db = new NorthwindBD();
public ActionResult Index() {
    try {
        var model = db.Categories.AsParallel().ToList();
        return View(model);
    }
    finally {
        if (db != null) db.Dispose();
    }
}

```

ListaProdutos é um método de controlador e contém o código que retorna os produtos de determinada categoria:

```

public ActionResult ListaProdutos(string categoria = "Beverages") {
    try {
        if (string.IsNullOrEmpty(categoria))
            return View("NotFound");
        ViewBag.Categoria = categoria;
        var model = (from p in db.Products
            where p.Category.CategoryName == categoria
            select p).ToList();
        if (model == null)
            return View("NotFound");
        else
            return PartialView("ViewProdutos", model);
    }
    finally {
        if (db != null) db.Dispose();
    }
}

```


Classe `AjaxOptions` contém as seguintes propriedades:

Propriedade	Descrição
<code>Confirm</code>	Exibe uma caixa de confirmação antes de postar as informações.
<code>HttpMethod</code>	Determina o tipo de operação. Exemplo: Get, Post.
<code>InsertionMode</code>	Define como a resposta será inserida na página. Exemplo: antes, após ou redefine o conteúdo atual.
<code>LoadingElementId</code>	Define o Id do elemento HTML executado, enquanto o Ajax está carregando.
<code>LoadingElementDuration</code>	Determina o tempo que o elemento definido na propriedade <code>LoadingElementId</code> permanece visível. Tempo é em milissegundos.
<code>OnBegin</code>	Define ou retorna a função JavaScript executada imediatamente antes de a página ser atualizada.
<code>OnComplete</code>	Define ou retorna a função JavaScript executada quando a operação assíncrona é completada.
<code>OnFailure</code>	Define ou retorna a função JavaScript executada quando a operação assíncrona falha.
<code>OnSuccess</code>	Define ou retorna a função JavaScript executada quando a operação assíncrona é completada com sucesso.
<code>UpdateTargetId</code>	Define o Id do elemento HTML usado para exibir a resposta enviada do servidor.
<code>Url</code>	Define ou retorna uma URL para usar na requisição.

No view `Index.cshtml` invocamos o método de controlador `ListaProdutos`:

```
<div id="divProdutos">
    @{Html.RenderAction("ListaProdutos");}
</div>
```

Exemplo:

```
@model IEnumerable<AppCapitulo10.Models.Category>
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
    <head>
        <title>Produtos por categoria</title>
        <script src="@Url.Content("~/Scripts/jquery-1.4.4.js")" type="text/javascript"></script>
        <script src="@Url.Content("~/Scripts/jquery.unobtrusive-ajax.js")" type="text/javascript">
        </script>
    </head>
    <body>
        <div>
            <ul style="display: inline">
                @foreach (var item in Model) {
```

```

<li style="display: inline">
    @Ajax.ActionLink(item.CategoryName, "ListaProdutos", new {
        categoria = item.CategoryName }, new AjaxOptions {
            UpdateTargetId = "divProdutos" })
</li>
}
</ul>
<hr />
<div id="divProdutos">
    @{Html.RenderAction("ListaProdutos");}
</div>
</div>
</body>
</html>

```

O partial view `ViewProdutos` contém o código responsável por exibir os produtos de cada categoria:

```

@model IEnumerable<AppCapitulo10.Models.Product>
<u><b>@ViewBag.Categoria</b></u></u>
<ul>
    @foreach (var item in Model) {
        <li>
            @item.ProductName
        </li>
    }
</ul>

```

Na figura 10.2 vemos o exemplo em ação:

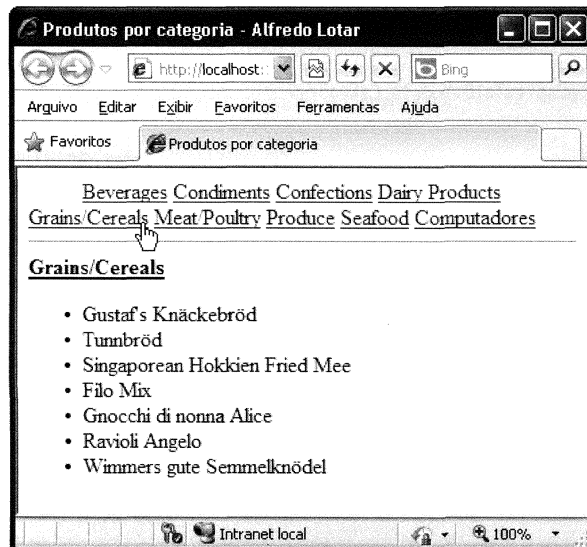


Figura 10.2 – Ajax em ação.

10.4 Enviando um formulário de forma assíncrona

O método `Ajax.BeginForm` posta o conteúdo de um formulário de forma assíncrona:

```
@using (Ajax.BeginForm("Find", "Home", new AjaxOptions {UpdateTargetId = "Resultado" })) { }
```

A propriedade `UpdateTargetId` contém o Id do elemento HTML usado para exibir a resposta enviada do servidor:

```
<div id="Resultado">  
-----  
</div>
```

Na tag `div` carregamos o partial view:

```
<div id="Resultado">  
    @{Html.RenderAction("Find");}  
</div>
```

O método `Find` carrega o partial view:

```
public ActionResult Find(string busca) {  
    IList<Product> model = new List<Product>();  
    // Restante do código  
    return PartialView("ViewProdutos", model);  
}
```

Na figura 10.3 temos um fluxograma que ilustra a estrutura de um mecanismo de busca de produtos.

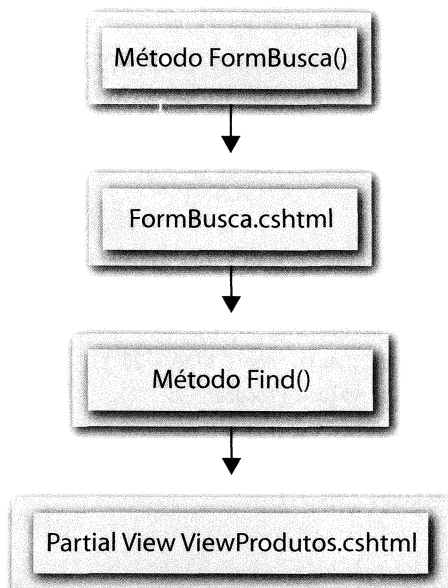


Figura 10.3 – Fluxograma de um mecanismo de busca.

No controlador `Home` temos o código do método `FormBusca` e `Find`:

```
public ActionResult FormBusca() {
    return View();
}
public ActionResult Find(string busca) {
    try {
        IList<Product> model = new List<Product>();
        if (!string.IsNullOrEmpty(busca)) {
            model = (from p in db.Products
                    where p.ProductName.StartsWith(busca)
                    select p).AsParallel().ToList();
            if (model == null)
                return View("NotFound");
        }
        return PartialView("ViewProdutos", model);
    }
    finally {
        if (db != null) db.Dispose();
    }
}
```

O arquivo `FormBusca.cshtml` contém o formulário e carrega o resultado da busca:

```
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
<title>Localizar produtos</title>
<script src="@Url.Content("~/Scripts/jquery-1.4.4.js")" type="text/javascript"></script>
<script src="@Url.Content("~/Scripts/jquery.unobtrusive-ajax.js")" type="text/javascript">
</script>
</head>
<body>
<div>
    @using (Ajax.BeginForm("Find", "Home", new AjaxOptions {
        UpdateTargetId = "Resultado" })) {
        @:Digite os termos da busca<br />
        @Html.TextBox("busca", null, new { size = 20 })
        <input type="submit" value="Procurar" />
    }
    <div id="Resultado">
        @{Html.RenderAction("Find");}
    </div>
</div>
</body>
</html>
```

O método Find carrega o partial view ViewProdutos.cshtml responsável pela exibição dos produtos:

```
@model IEnumerable<AppCapítulo10.Models.Product>
<u><b>@ViewBag.Categoria</b></u>
<ul>
    @foreach (var item in Model) {
        <li>
            @item.ProductName
        </li>
    }
</ul>
```

Na figura 10.4 vemos o mecanismo de busca em ação:

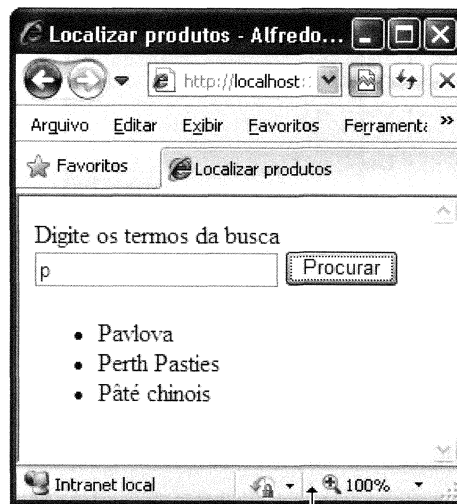


Figura 10.4 – Mecanismo de busca em ação.

10.5 Mensagens de confirmação

Para o método Ajax.BeginForm temos a opção de definir mensagens de confirmação. Útil, por exemplo, quando o usuário está prestes a efetuar tarefas que merecem atenção, como excluir um registro do banco de dados.

Exemplo:

```
@model IEnumerable<AppCapítulo10.Models.Product>
<ul>
    @foreach (var item in Model) {
        <li>
            @item.ProductName
            @Ajax.ActionLink("Excluir", "", new AjaxOptions {
```

```

        Confirm = "Você confirma esta ação?" })
    </li>
}
</ul>

```

Observe a figura 10.5:

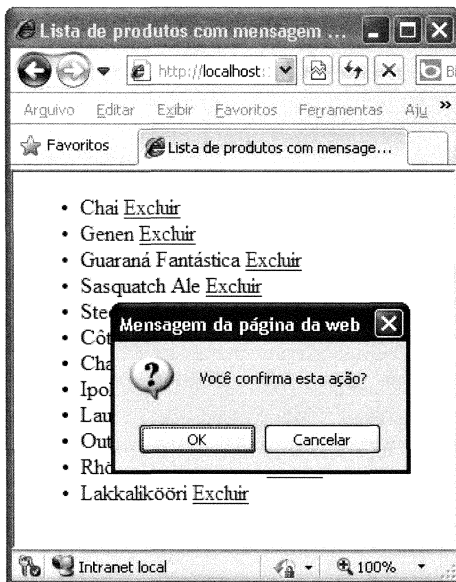


Figura 10.5 – Mensagem de confirmação.

10.6 Propriedade OnSuccess

A propriedade `OnSuccess` define ou retorna a função JavaScript executada quando a operação assíncrona é completada com sucesso.

Exemplo:

```

<script type="text/javascript">
    function Mensagem() {
        $('p').text('Transação concluída com sucesso.');
```

 }
</script>
@using (Ajax.BeginForm("Index", "Home", new AjaxOptions { OnSuccess = "Mensagem" })) {
 // Código aqui
}
<p></p>

10.7 Acrescentando elementos visuais

A propriedade `LoadingElementId` e `LoadingElementDuration` da classe `AjaxOptions` definem, respectivamente, o Id do elemento HTML executado, enquanto o Ajax está carregando e o tempo que o elemento definido na propriedade `LoadingElementId` permanece visível.

A implementação é simples. Basta definir a propriedade `LoadingElementId` e, se necessário, também a propriedade `LoadingElementDuration` no método `ActionLink` ou `BeginForm`.

Exemplo:

```
@Ajax.ActionLink(item.CategoryName, "ListaProdutos", new { categoria = item.CategoryName },
    new AjaxOptions { LoadingElementId = "divLoad", LoadingElementDuration = 10})
```

A tag `div` exibe o elemento visual, como um texto:

```
<div id="divLoad" style="display: none">
    <p>Carregando ... </p>
</div>
```

Ou uma figura:

```
<div id="divLoad" style="display: none">
    <image src="../../Content/imagem.jpg" alt="" />
</div>
```

10.8 Método `IsAjaxRequest`

Verifica se a requisição HTTP é uma requisição Ajax.

Exemplo:

```
public ActionResult ListaProdutos(string categoria = "Beverages") {
    // Código aqui
    var model = (from p in db.Products
        where p.Category.CategoryName == categoria
        select p).ToList();
    if (!Request.IsAjaxRequest()) {
        return View("NoAjax", model);
    }
    else {
        return PartialView("ViewProdutos", model);
    }
}
```

Segurança de aplicações ASP.NET MVC

A segurança de uma aplicação é muito importante em um website, pois centenas, senão milhares de usuários mal-intencionados tentarão executar alguma ação maliciosa na sua aplicação. Por isso, é importante que você valide todas as entradas, divida a aplicação em áreas públicas e restritas, use SSL e CAPTCHAs, criptografe informações sigilosas, permita somente senhas fortes.

Nunca faça nada sério na Internet; digamos assim, se o website não é seguro.

O assunto segurança de websites é bastante extenso e complexo; então, se você precisa de um website realmente seguro para comercializar produtos, serviços etc., contrate empresas especializadas, consultores independentes ou use softwares para encontrar brechas de segurança na rede, no servidor web, no servidor de banco de dados e no código da aplicação. Mesmo que você tenha um conhecimento profundo de segurança de websites e seja bastante cuidadoso e organizado, é provável que o seu software tenha alguma brecha de segurança ainda não descoberta. Os softwares listados a seguir podem ajudá-lo a se proteger:

- <http://www.acunetix.com/vulnerability-scanner/download.htm>
- <http://www.microsoft.com/download/en/details.aspx?id=1677>
- <http://www8.hp.com/us/en/software/software-solution.html?compURI=tcm:245-936139#tab=1>
- <http://www-01.ibm.com/software/awdtools/appscan/>
- <https://www.siteblindado.com/pt>

11.1 Criando uma nova aplicação

Para testar os exemplos deste capítulo, inicie o Visual Studio. Acesse o menu **File** e clique em **New Project**

Na caixa de diálogo **New Project**, selecione **Web > Visual Studio 2012 > ASP.NET MVC 4 Web Application**. Nomeie o projeto como `AppCapitulo11`. Clique em **OK**.

Em seguida, na caixa de combinação **View Engine**, selecione o mecanismo de exibição **Razor**. Selecione o template **Internet Application**. Observe a figura 11.1:

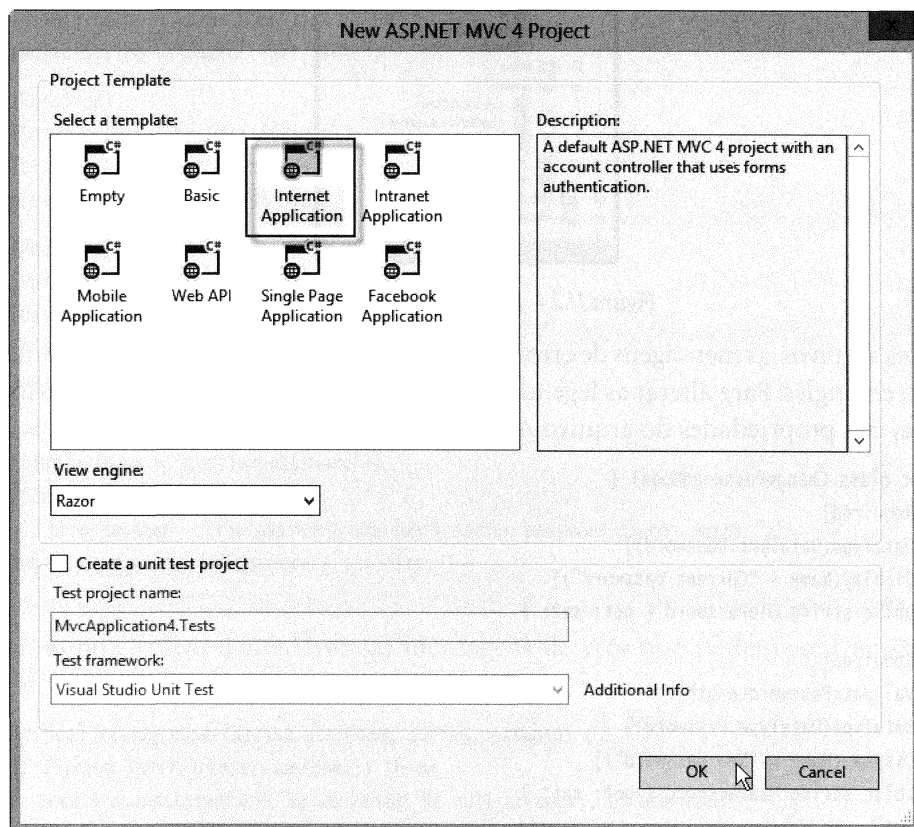


Figura 11.1 – Caixa de diálogo New ASP.NET MVC 4 Project.

Quando optamos pelo template **Internet Application**, são acrescentados ao projeto automaticamente arquivos responsáveis por manipular informações de autenticação de usuários, como cadastro de usuários, alteração de senhas e o login propriamente dito. Observe a figura 11.2:

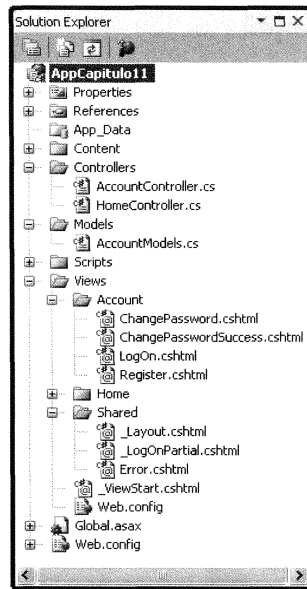


Figura 11.2 – Janela Solution Explorer.

Nestes arquivos, as mensagens de erro, o título e as legendas dos campos de formulário estão em inglês. Para alterar as legendas dos campos de formulário defina o atributo `Display` nas propriedades do arquivo `AccountModels.cs`:

```
public class ChangePasswordModel {
    [Required]
    [DataType(DataType.Password)]
    [Display(Name = "Current password")]
    public string OldPassword { get; set; }

    [Required]
    [ValidatePasswordLength]
    [DataType(DataType.Password)]
    [Display(Name = "New password")]
    public string NewPassword { get; set; }

    [DataType(DataType.Password)]
    [Display(Name = "Confirm new password")]
    [Compare("NewPassword",
        ErrorMessage = "The new password and confirmation password do not match.")]
    public string ConfirmPassword { get; set; }
}

public class LogOnModel {
    [Required]
    [Display(Name = "User name")]
    public string UserName { get; set; }
```

```

    [Required]
    [DataType(DataType.Password)]
    [Display(Name = "Password")]
    public string Password { get; set; }

    [Display(Name = "Remember me?")]
    public bool RememberMe { get; set; }
}

public class RegisterModel {
    [Required]
    [Display(Name = "User name")]
    public string UserName { get; set; }

    [Required]
    [DataType(DataType.EmailAddress)]
    [Display(Name = "Email address")]
    public string Email { get; set; }

    [Required]
    [ValidatePasswordLength]
    [DataType(DataType.Password)]
    [Display(Name = "Password")]
    public string Password { get; set; }

    [DataType(DataType.Password)]
    [Display(Name = "Confirm password")]
    [Compare("Password",
        ErrorMessage = "The password and confirmation password do not match.")]
    public string ConfirmPassword { get; set; }
}

```

No mesmo arquivo temos diversas mensagens de erro que podem ser traduzidas. Observe o código a seguir:

```

public bool ValidateUser(string userName, string password) {
    if (String.IsNullOrEmpty(userName)) throw
        new ArgumentException("Value cannot be null or empty.", "userName");
    if (String.IsNullOrEmpty(password)) throw
        new ArgumentException("Value cannot be null or empty.", "password");

    return _provider.ValidateUser(userName, password);
}

public MembershipCreateStatus CreateUser(string userName, string password, string email) {
    if (String.IsNullOrEmpty(userName)) throw
        new ArgumentException("Value cannot be null or empty.", "userName");
    if (String.IsNullOrEmpty(password)) throw
        new ArgumentException("Value cannot be null or empty.", "password");
}

```

```

        if (String.IsNullOrEmpty(email)) throw
            new ArgumentException("Value cannot be null or empty.", "email");

        MembershipCreateStatus status;
        _provider.CreateUser(userName, password, email, null, null, true, null, out status);
        return status;
    }

    public bool ChangePassword(string userName, string oldPassword, string newPassword) {
        if (String.IsNullOrEmpty(userName)) throw
            new ArgumentException("Value cannot be null or empty.", "userName");
        if (String.IsNullOrEmpty(oldPassword)) throw
            new ArgumentException("Value cannot be null or empty.", "oldPassword");
        if (String.IsNullOrEmpty(newPassword)) throw
            new ArgumentException("Value cannot be null or empty.", "newPassword");
        try {
            MembershipUser currentUser = _provider.GetUser(userName, true /* userIsOnline */);
            return currentUser.ChangePassword(oldPassword, newPassword);
        }
        catch (ArgumentException) {
            return false;
        }
        catch (MembershipPasswordException) {
            return false;
        }
    }
}

public static class AccountValidation {
    public static string ErrorCodeToString(MembershipCreateStatus createStatus) {
        switch (createStatus) {
            case MembershipCreateStatus.DuplicateUserName:
                return "Username already exists. Please enter a different user name.";

            case MembershipCreateStatus.DuplicateEmail:
                return "A username for that e-mail address already exists. Please enter a
different e-mail address.";

            case MembershipCreateStatus.InvalidPassword:
                return "The password provided is invalid. Please enter a valid password value.";

            case MembershipCreateStatus.InvalidEmail:
                return "The e-mail address provided is invalid. Please check the value and try again.";

            case MembershipCreateStatus.InvalidAnswer:
                return "The password retrieval answer provided is invalid. Please check the
value and try again.";

            case MembershipCreateStatus.InvalidQuestion:
                return "The password retrieval question provided is invalid. Please check the
value and try again.";
        }
    }
}

```

```
case MembershipCreateStatus.InvalidUserName:
    return "The user name provided is invalid. Please check the value and try again.";

case MembershipCreateStatus.ProviderError:
    return "The authentication provider returned an error. Please verify your entry
and try again. If the problem persists, please contact your system administrator.";

case MembershipCreateStatus.UserRejected:
    return "The user creation request has been canceled. Please verify your entry
and try again. If the problem persists, please contact your system administrator.";

default:
    return "An unknown error occurred. Please verify your entry and try again. If
the problem persists, please contact your system administrator.";
}
}
```

Obs.: mensagens específicas de cada formulário podem ser alteradas acessando os arquivo .cshtml ou .aspx no diretório Views\Account.

A figura 11.3 ilustra o formulário de login traduzido para o português.

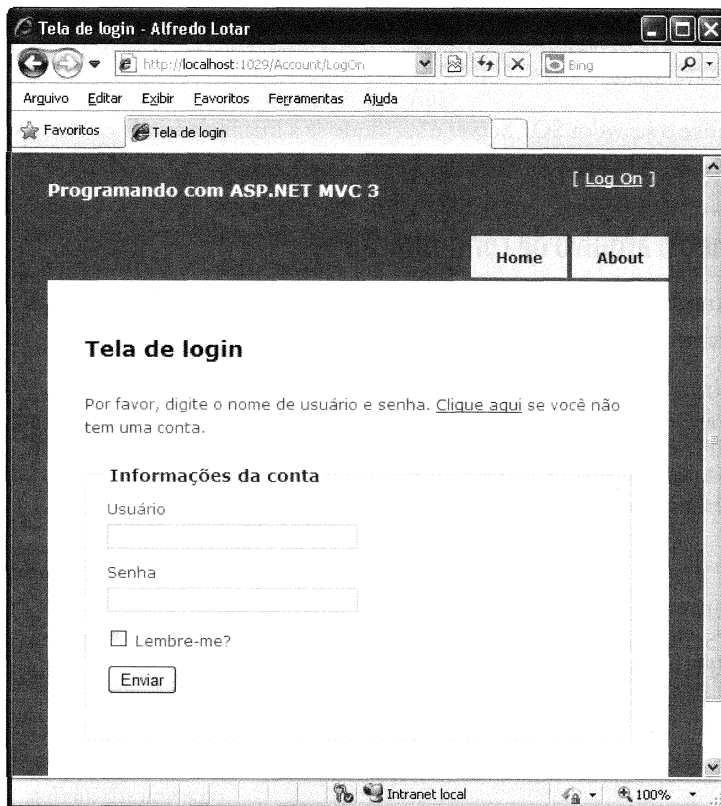


Figura 11.3 – Tela de login traduzida.

11.2 Criando o banco de dados de segurança

Para que você possa usar o mecanismo de autenticação adicionado ao projeto do ASP.NET MVC é necessário acrescentar um banco de dados ou as tabelas a um banco de dados existente.

O banco de dados contém as regras de acesso, os grupos de usuários e o próprio cadastro desses usuários.

Para criar o banco de dados no SQL Server, clique no menu **Iniciar > Todos os programas > Microsoft Visual Studio > Visual Studio Tools > Developer Command Prompt for VS2013**. Em seguida, digite:

```
aspnet_regsql.exe -S .\SQLEXPRESS -E -d NomeBancoDados -A mr
```

Exemplo:

```
aspnet_regsql.exe -S .\SQLEXPRESS -E -d Northwind -A mr
```

Se preferir, use o assistente:

```
aspnet_regsql.exe -W
```

Em ambos os casos são acrescentadas as seguintes tabelas ao banco de dados Northwind:

```
aspnet_Membership, aspnet_Roles, aspnet_Users, aspnet_UsersInRoles, aspnet_Applications,  
aspnet_SchemaVersions.
```

Obs.: ative o servidor SQL Server e verifique se a instância usada é .\SQLEXPRESS antes de usar a ferramenta aspnet_regsql.exe.

11.3 Alterando o arquivo de configuração

Modifique o arquivo de configuração web.config localizado na raiz da aplicação. Altere o elemento connectionStrings. Substitua:

```
<connectionStrings>  
  <add name="ApplicationServices" connectionString="data source=.\SQLEXPRESS;Integrated  
    Security=SSPI;AttachDBFilename=|DataDirectory|aspnetdb.mdf;User Instance=true"  
    providerName="System.Data.SqlClient" />  
</connectionStrings>
```

Por:

```
<connectionStrings>  
  <add name="SqlBDseguro" connectionString="Data Source=.\SQLEXPRESS;Integrated  
    Security=SSPI;Initial Catalog=Northwind;" />  
</connectionStrings>
```

Acrescente o elemento `membership`:

```
<membership defaultProvider="SqlProvider" >
  <providers>
    <clear />
    <add
      name="SqlProvider"
      type="System.Web.Security.SqlMembershipProvider"
      connectionStringName="SqlBDseguro"
      applicationName="AppCapitulo11"
      enablePasswordRetrieval="false"
      requiresUniqueEmail="true"
      minRequiredPasswordLength="8"
      minRequiredNonalphanumericCharacters="1"/>
    </providers>
  </membership>
```

A seguir, listamos os principais atributos do elemento-filho `Add`:

Atributo	Descrição
<code>name</code>	Nome do provedor de dados usados.
<code>type</code>	Refere-se à classe que implementa a classe abstrata <code>MembershipProvider</code> . Exemplo: <code>SqlMembershipProvider</code> , <code>ActiveDirectoryMembershipProvider</code> .
<code>connectionStringName</code>	Define a string de conexão usada para se conectar ao banco de dados. Essa string é definida no elemento <code>connectionStrings</code> .
<code>applicationName</code>	Define o nome da aplicação e deve ser definida para evitar eventuais problemas de identificação ao migrar a aplicação de um servidor para outro. Ou seja, você precisa manter o mesmo nome ao migrar a aplicação de um servidor para outro ou você perde acesso aos usuários e regras de acesso previamente cadastrados.
<code>enablePasswordRetrieval</code>	É permitida a recuperação da senha pelo usuário? Sim ou não?
<code>requiresUniqueEmail</code>	Cada usuário deverá informar um e-mail único? Sim ou não?
<code>minRequiredPasswordLength</code>	Determina o comprimento mínimo da senha. O valor-padrão é 7. Recomendamos valores maiores que o padrão, no mínimo 8. No meu computador uso no mínimo 14.
<code>MinRequiredNonalphanumericCharacters</code>	Número mínimo de caracteres não alfanuméricos que a senha deve conter. O valor-padrão é 1.
<code>maxInvalidPasswordAttempts</code>	Número máximo de tentativas inválidas de login antes da conta ser bloqueada. O valor-padrão é 5.

Em seguida, acrescentamos ao arquivo `web.config` o elemento `roleManager` usado para gerenciar as regras de acesso:

```

<roleManager defaultProvider="SqlRoleProvider"
    enabled="true"
    cacheRolesInCookie="true"
    cookieName="{278FAD2B-26C2-4940-932B-4780E200569B}"
    cookieTimeout="5"
    cookieRequireSSL="true"
    cookieSlidingExpiration="true"
    cookieProtection="All"
    createPersistentCookie="false">
    <providers>
        <add
            name="SqlRoleProvider"
            type="System.Web.Security.SqlRoleProvider"
            connectionStringName="SqlBDseguro"
            applicationName="AppCapitulo11"/>
    </providers>
</roleManager>

```

Os principais atributos do elemento `roleManager` são:

Atributo	Descrição
<code>defaultProvider</code>	Nome do provedor de dados usados.
<code>enabled</code>	Ativa ou desativa o gerenciador de regras de acesso.
<code>cacheRolesInCookie</code>	Coloca em um cookie os grupos de usuários ao qual o usuário logado pertence.
<code>cookieName</code>	Determina o nome do cookie. O valor-padrão é <code>.ASPXROLES</code> .
<code>cookieTimeout</code>	Tempo de vida do cookie. O valor-padrão é de 30 minutos. É interessante que você use um valor inferior ao padrão.
<code>cookieRequireSSL</code>	Ative sempre SSL para cookies de autenticação. Defina este atributo sempre como <code>true</code> .
<code>cookieSlidingExpiration</code>	O tempo do cookie é renovado a cada nova requisição? Sim ou não?
<code>cookieProtection</code>	Defina este atributo sempre como <code>All</code> . Significa que o cookie é armazenado criptografado e é a prova de alterações.
<code>createPersistentCookie</code>	Valor-padrão é <code>false</code> e deve ser mantido assim. Por motivos de segurança não recomendamos manter cookies por longo período de tempo.

Obs.: não use o valor-padrão no atributo `cookieName` quando múltiplos websites estiverem hospedados no mesmo servidor. Nesse caso, defina um GUID como nome. Clique no menu **Iniciar > Todos os programas > Microsoft Visual Studio 2010 > Microsoft Windows SDK Tools > GUID Generator**. Na janela **Create GUID**, gere um GUID no formato registro e copie para o atributo `cookieName`.

11.4 Gerenciando a aplicação

Para ilustrar este tópico, acrescentamos a aplicação à área Admin. Consulte o tópico 6.4.

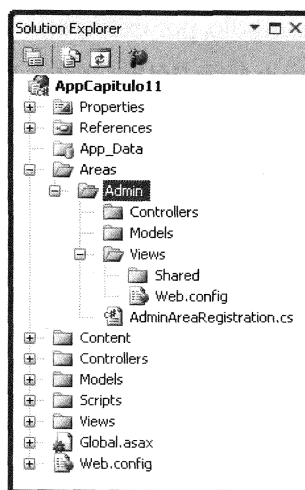


Figura 11.4 – Área Admin.

Após a criação da área, acrescentamos as tabelas ao banco de dados SQL Server, criamos os diretórios e configuramos os elementos `membership` e `roleManager` no arquivo `web.config`. Nesse ponto, estamos prontos para adicionar usuários, regras de acesso e grupos de usuários:

- No Visual Studio, acesse menu **Web site > ASP.NET Configuration**. Clique na aba **Segurança (Security)** e, em seguida, em **Usar o Assistente para Configuração da Segurança a fim de configurar a segurança, passo a passo (Use the security Setup Wizard to configure security step by step)**.
- O assistente lhe dá as boas-vindas e, para continuar, clique em **Avançar (Next)**.
- Selecione o método de acesso Da Internet ou De uma rede local (From a local area network). Selecione a opção **Da Internet (From the internet)**, a qual insere a seguinte linha no `web.config`:

```
<authentication mode="Forms"/>
```

Obs.: o ASP.NET suporta três tipos de autenticação: Windows, Forms e Passport.

- Na tela **Configurações avançadas do provedor (Advanced provider settings)**, clique no botão com a legenda **Avançar (Next)**.
- Na tela **Definir Funções (Define Roles)**, marque a opção: **Habilita as funções para este site (Enable roles for this Web site)**. Em seguida, clique em **Avançar (Next)**.
- Adicione novas funções (grupos de usuários) à aplicação. Para testar esse exemplo, adicione as funções Administradores e Clientes. Depois de adicionar as funções, clique no botão **Avançar (Next)**.

- Adicione novos usuários, pois, para testar esse exemplo, recomendamos, no mínimo, dois deles. Depois de adicioná-los, clique no botão **Avançar (Next)**. Atenção: a senha deve ter o comprimento mínimo de 8 caracteres e um caractere não alfanumérico (um símbolo, por exemplo).
- Na tela **Adicionar Novas Regras de Acesso (Add New Access Rules)**, adicione regras para controlar o acesso a todo o site ou a pastas individuais. Selecione a pasta **Admin** e, em seguida, selecione a função **Administradores**. Selecione **Permitir (Allow)**; clique no botão **Adicionar esta Regra (Add This Rule)**. Repita a operação para os demais usuários.
- Clique em **Concluir (Finish)**. Após essa etapa, temos um novo arquivo de configuração `web.config` no diretório `Admin`:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.web>
    <authorization>
      <allow roles="Administradores" />
    </authorization>
  </system.web>
</configuration>
```

- Acrescente a linha a seguir ao arquivo `web.config` localizado no diretório `Admin`:

```
<deny users="*" />
```

Exemplo:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.web>
    <authorization>
      <allow roles="Administradores" />
      <deny users="*" />
    </authorization>
  </system.web>
</configuration>
```

- Na aba **Segurança (Security)**, clique em **Gerenciar usuários (Manage users)**. Na lista de usuários ativos, clique em **Editar funções (Edit roles)**. Marque a caixa de seleção **Administradores** para um dos usuários.

Você pode acrescentar também usuários via código C#:

```
MembershipCreateStatus status;
MembershipUser novoUsuario = Membership.CreateUser("Usuário", "Senha",
    "email@teste.com.br", "Pergunta", "Resposta", true, out status);
if (novoUsuario == null) {
    // status;
}
```

```

else {
    // Usuário adicionado com sucesso
}

```

Inclusive adicionar o usuário ao grupo:

```
Roles.AddUserToRole("NomeUsuário", "Grupo");
```

11.4.1 A aplicação

A aplicação usada neste capítulo é extremamente simples. Temos uma tela que exibe as categorias e outra tela, os produtos. As categorias são exibidas por intermédio do método de controlador `Index`:

```

public ActionResult Index() {
    try {
        var model = db.Categories.AsParallel().ToList();
        if (model == null)
            return View("NotFound");
        else
            return View(model);
    }
    finally {
        if (db != null) db.Dispose();
    }
}

```

E o view `Index.cshtml`:

```

@model IEnumerable<AppCapitulo11.Models.Category>
@{
    ViewBag.Title = "Lista de categorias";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
<ul>
    @foreach (var item in Model) {
        <li>@Html.ActionLink(item.CategoryName, "Details", new { categoria = item.
            CategoryName })</li>
    }
}
</ul>

```

Enquanto os produtos são exibidos por meio do método de controlador `Details`:

```

public ActionResult Details(string categoria) {
    try {
        ViewBag.Message = categoria;
        var model = (from p in db.Products
            where p.Category.CategoryName == categoria
            select p).ToList();
        if (model == null)
            return View("NotFound");
    }
}

```

```

        else
            return View(model);
    }
    finally {
        if (db != null) db.Dispose();
    }
}

```

E pelo view Details.cshtml:

```

@model IEnumerable<AppCapitulo11.Models.Product>
@{
    ViewBag.Title = ViewBag.Message;
    Layout = "~/Views/Shared/_Layout.cshtml";
}
<h2>@ViewBag.Message</h2>
<ul>
    @foreach (var item in Model) {
        <li>@item.ProductName</li>
    }
}
</ul>

```

Os recursos protegidos são dois formulários, um de inserção e outro de atualização, ambos criados na área Admin. Para acrescentar uma nova categoria, a aplicação usa o método Create:

```

public ActionResult Create() {
    return View();
}
[HttpPost]
public ActionResult Create(Category entidade) {
    try {
        if (!ModelState.IsValid) {
            ModelState.AddModelError("",
                "Ocorreram erros no formulário. Por favor, corrija os erros e tente novamente.");
            return View();
        }
        else {
            db.AddToCategories(entidade);
            db.SaveChanges();
            return RedirectToAction("Index");
        }
    }
    catch (System.Data.EntitySqlException) {
        ModelState.AddModelError("", "Erro EntitySqlException.");
        return View();
    }
    catch (System.Data.EntityCommandExecutionException) {
        ModelState.AddModelError("", "Erro EntityCommandExecutionException.");
        return View();
    }
}

```

```

    finally {
        if (db != null) db.Dispose();
    }
}

```

E um view com o mesmo nome:

```

@model AppCapitulo11.Models.Category
@{
    ViewBag.Title = "Create";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
<h2>Create</h2>
<script src="@Url.Content("~/Scripts/jquery.validate.min.js")" type="text/javascript"></script>
<script src="@Url.Content("~/Scripts/jquery.validate.unobtrusive.min.js")" type="text/javascript">
</script>
@using (Html.BeginForm()) {
    @Html.ValidationSummary(true)
    <fieldset>
        <legend>Nova categoria</legend>
        <div class="editor-label">
            @Html.LabelFor(model => model.CategoryName)
        </div>
        <div class="editor-field">
            @Html.TextBoxFor(model => model.CategoryName, new { @style = "width: 250px;" })
            @Html.ValidationMessageFor(model => model.CategoryName)
        </div>
        <div class="editor-label">
            @Html.LabelFor(model => model.Description)
        </div>
        <div class="editor-field">
            @Html.TextAreaFor(model => model.Description, new { @style = "width: 250px;" })
            @Html.ValidationMessageFor(model => model.Description)
        </div>
        <p>
            <input type="submit" value="Create" />
        </p>
    </fieldset>
}
<div>
    @Html.ActionLink("Página inicial", "Index")
</div>

```

A atualização é realizada com o método Edit:

```

public ActionResult Edit(int? id) {
    try {
        if (!id.HasValue)
            return View("NotFound");
        var model = db.Categories.Where(c => c.CategoryID == id).FirstOrDefault();
        if (model == null)
            return View("NotFound");
    }
}

```

```

        else
            return View(model);
    }
    finally {
        if (db != null) db.Dispose();
    }
}
[HttpPost]
public ActionResult Edit(Category entidade) {
    try {
        var model = db.Categories.Where(c => c.CategoryID == entidade.CategoryID).
            FirstOrDefault();
        if (model == null) {
            return View("NotFound");
        }
        else {
            db.ApplyCurrentValues(model.EntityKey.EntitySetName, entidade);
            db.SaveChanges();
            return RedirectToAction("Index");
        }
    }
    catch (System.Data.EntitySqlException) {
        ModelState.AddModelError("", "Erro EntitySqlException.");
        return View();
    }
    catch (System.Data.EntityCommandExecutionException) {
        ModelState.AddModelError("", "Erro EntityCommandExecutionException.");
        return View();
    }
    finally {
        if (db != null) db.Dispose();
    }
}
}

```

E o view Edit.cshtml:

```

@model AppCapitulo11.Models.Category
@{
    ViewBag.Title = "Edit";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
<script src="@Url.Content("~/Scripts/jquery.validate.min.js")" type="text/javascript"></script>
<script src="@Url.Content("~/Scripts/jquery.validate.unobtrusive.min.js")" type="text/javascript">
</script>
@using (Html.BeginForm("Edit", "Home")) {
    @Html.ValidationSummary(true)
    <fieldset>
        <legend>Editar categoria</legend>
        @Html.HiddenFor(model => model.CategoryID)
        <div class="editor-label">
            @Html.LabelFor(model => model.CategoryName)
        </div>
    </fieldset>
}

```

```

        <div class="editor-field">
            @Html.TextBoxFor(model => model.CategoryName, new { @style = "width: 250px;" })
            @Html.ValidationMessageFor(model => model.CategoryName)
        </div>
        <div class="editor-label">
            @Html.LabelFor(model => model.Description)
        </div>
        <div class="editor-field">
            @Html.TextAreaFor(model => model.Description, new { @style = "width: 250px;" })
            @Html.ValidationMessageFor(model => model.Description)
        </div>
        <p>
            <input type="submit" value="Save" />
        </p>
    </fieldset>
}
</div>
    @Html.ActionLink("Página inicial", "Index")
</div>

```

Para evitar conflitos entre os controladores com o mesmo nome, acrescentamos a linha a seguir:

```
new[] { "AppCapítulo11.Controllers" }
```

Ao arquivo Global.asax:

```

public static void RegisterRoutes(RouteCollection routes) {
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
    routes.MapRoute(
        "Default",
        "{controller}/{action}/{id}",
        new { controller = "Home", action = "Index", id = UrlParameter.Optional },
        new[] { "AppCapítulo11.Controllers" }
    );
}

```

E também alteramos os links da página de layout.

Exemplo:

```
@Html.ActionLink("Home", "Index", "Home", new { area = string.Empty }, null)
```

11.5 ADO.NET Entity Framework

Acrescente a aplicação às classes do ADO.NET Entity Framework, conforme o tópico 4.3. Em seguida, adicione o arquivo Categoria.cs e a classe Category:

```

using System;
using System.Collections.Generic;
using System.Linq;

```

```
using System.Web;

namespace AppCapitulo11.Models {
    public class Category {
    }
}
```

Realize a formatação e validação dos campos:

```
using System.ComponentModel.DataAnnotations;
namespace AppCapitulo11.Models {
    [MetadataType(typeof(CategoryMetadata))]
    public partial class Category { }

    public class CategoryMetadata {
        [Required(ErrorMessage = "O nome da categoria é obrigatório.")]
        [StringLength(15, MinimumLength = 3,
            ErrorMessage = "Digite no mínimo 3 e no máximo 15 caracteres.")]
        [RegularExpression(@"^[a-zA-Z0-9_\-\/\s]{3,15}$",
            ErrorMessage="Digite uma categoria válida.")]
        [DisplayName = "Categoria"]
        public string CategoryName { get; set; }

        [StringLength(150, MinimumLength = 0,
            ErrorMessage = "Digite no máximo 150 caracteres.")]
        [DisplayName = "Descrição"]
        public string Description { get; set; }
    }
}
```

O atributo `RegularExpression` valida a propriedade por meio de uma expressão regular:

```
[RegularExpression(@"^[a-zA-Z0-9_\-\/\s]{3,15}$", ErrorMessage="Digite uma categoria válida.")]
```

Validação centralizada é recomendada. Assim como a validação em múltiplas camadas, valide também as entradas dos campos de formulário no método de controlador:

```
[HttpPost]
public ActionResult Create(Category entidade) {
    try {
        if (string.IsNullOrEmpty(entidade.CategoryName)
            || !Regex.IsMatch(entidade.CategoryName, @"^[a-zA-Z0-9_\-\/\s]{3,15}$"))
            ModelState.AddModelError("CategoryName", "Digite uma categoria válida.");

        if (string.IsNullOrEmpty(entidade.Description)
            || !Regex.IsMatch(entidade.Description, @"^[0-9a-zA-Z_\.\/\s]{0,150}$"))
            ModelState.AddModelError("Description", "Caracteres inválidos na descrição.");

        if (!ModelState.IsValid) {
            ModelState.AddModelError("",
```



```

        "Ocorreram erros no formulário. Por favor, corrija os erros e tente novamente.");
        return View();
    }
    // Restante do código
}
// Restante do código
}

```

Obs.: garanta que a entrada contenha somente o conteúdo desejado. Utilize o acento circunflexo (^) e um cifrão (\$) como delimitadores no início e no fim de expressões.

11.5.1 Validando as propriedades de autenticação

Valide as propriedades de autenticação localizadas no arquivo `AccountModels.cs`:

Exemplo:

```

public class ChangePasswordModel {
    [Required(ErrorMessage="Senha é obrigatória.")]
    [DataType(DataType.Password)]
    [RegularExpression(@"^[a-zA-Z0-9_\-\.\\?\!]{8,128}$",
        ErrorMessage = "Digite uma senha válida. Use letras, números e os caracteres: *-+.?!")]
    [Display(Name = "Senha atual")]
    public string OldPassword { get; set; }

    [Required(ErrorMessage = "Nova senha é obrigatória.")]
    [ValidatePasswordLength]
    [DataType(DataType.Password)]
    [RegularExpression(@"^[a-zA-Z0-9_\-\.\\?\!]{8,128}$",
        ErrorMessage = "Digite uma senha válida. Use letras, números e os caracteres: *-+.?!")]
    [Display(Name = "Nova senha")]
    public string NewPassword { get; set; }

    [Required(ErrorMessage = "Senha de confirmação é obrigatória.")]
    [DataType(DataType.Password)]
    [Display(Name = "Confirme a senha")]
    [Compare("NewPassword", ErrorMessage = "A nova senha não coincide com a senha de
    confirmação. Por favor, digite novamente.")]
    public string ConfirmPassword { get; set; }
}

public class LogOnModel {
    [Required(ErrorMessage = "Nome do usuário é obrigatório.")]
    [RegularExpression(@"^[a-zA-Z0-9]{1,256}$",
        ErrorMessage = "Digite um nome de usuário válido.")]
    [Display(Name = "Usuário")]
    public string UserName { get; set; }
}

```

```

[Required(ErrorMessage = "Senha é obrigatória.")]
[DataType(DataType.Password)]
[RegularExpression(@"^[a-zA-Z0-9_\-\.\\?\!]{8,128}$",
    ErrorMessage = "Digite uma senha válida. Use letras, números e os caracteres: *-+.?!")]
[Display(Name = "Senha")]
public string Password { get; set; }

[Display(Name = "Lembre-me?")]
public bool RememberMe { get; set; }
}

public class RegisterModel {
    [Required(ErrorMessage = "Nome do usuário é obrigatório.")]
    [RegularExpression(@"^[a-zA-Z0-9]{1,256}$", ErrorMessage = "Digite um nome de usuário válido. Não use espaços. Somente letra e números.")]
    [Display(Name = "Usuário")]
    public string UserName { get; set; }

    [Required(ErrorMessage = "E-mail é obrigatório.")]
    [DataType(DataType.EmailAddress)]
    [RegularExpression(@"^[w+([-+.]\w+)*@[w+([-+]\w+)*\.[w+([-+]\w+)*$",
        ErrorMessage = "Digite um e-mail válido.")]
    [Display(Name = "E-mail")]
    public string Email { get; set; }

    [Required(ErrorMessage = "Senha é obrigatória.")]
    [ValidatePasswordLength]
    [RegularExpression(@"^[a-zA-Z0-9_\-\.\\?\!]{8,128}$", ErrorMessage = "Digite uma senha válida. Use letras, números e os caracteres: *-+.?!")]
    [DataType(DataType.Password)]
    [Display(Name = "Senha")]
    public string Password { get; set; }

    [Required(ErrorMessage = "Senha de confirmação é obrigatória.")]
    [DataType(DataType.Password)]
    [RegularExpression(@"^[a-zA-Z0-9_\-\.\\?\!]{8,128}$",
        ErrorMessage = "Digite uma senha válida. Use letras, números e os caracteres: *-+.?!")]
    [Display(Name = "Confirme a senha")]
    [Compare("Password", ErrorMessage = "A nova senha não coincide com a senha de confirmação. Por favor, digite novamente.")]
    public string ConfirmPassword { get; set; }
}

```

No arquivo de configuração `web.config` você pode definir o atributo `passwordStrengthRegularExpression` e determinar uma expressão regular para avaliar a senha.

Exemplo:

```

<membership defaultProvider="SqlProvider">
  <providers>
    <clear />
    <add name="SqlProvider" type="System.Web.Security.SqlMembershipProvider"
      connectionStringName="SqlBDseguro" applicationName="AppCapitulo11"
      enablePasswordRetrieval="false" requiresUniqueEmail="true"
      minRequiredPasswordLength="8" minRequiredNonalphanumericCharacters="1"
      passwordStrengthRegularExpression="^[a-zA-Z0-9_\\*\\-\\.\\?\\!]{8,15}$" />
  </providers>
</membership>

```

Obs.: as expressões regulares usadas nos exemplos deste livro devem ser analisadas e testadas antes de serem implementadas em aplicações reais.

11.6 Segurança de acesso ao código

Você precisa restringir o acesso aos recursos protegidos, ou seja, não basta ter um formulário de login; é preciso determinar quem pode acessar o quê.

O princípio da defesa em profundidade rege a utilização de várias camadas de proteção. Você deve restringir o acesso a um recurso por intermédio do arquivo de configuração web.config:

```

<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.web>
    <authorization>
      <allow roles="Administradores" />
      <deny users="*/>
    </authorization>
  </system.web>
</configuration>

```

No método de controlador com o atributo `Authorize`:

```

[HttpPost, Authorize(Roles = "Administradores")]
public ActionResult Create(Category entidade) {
    // Restante do código
}

```

E por meio de segurança de acesso ao código. Nesse caso, aplicamos o atributo `PrincipalPermission` à classe parcial `Category` do ADO.NET Entity Framework:

```

[PrincipalPermission(SecurityAction.Demand, Role = "Administradores")]
public void AddToCategories(Category category) {
    base.AddObject("Categories", category);
}

```

Obs.: se você alterar as classes do ADO.NET Entity Framework com o assistente do Visual Studio, as alterações personalizadas serão perdidas.

Mesmo com todos esses cuidados, um componente ainda pode ser acessado por um usuário mal-intencionado. Para evitar que alguém use o seu componente de acesso a dados, assine o componente com chaves criptografadas. No Developer Command Prompt for VS2013, digite:

```
sn -k chaves.snk
```

Copie o arquivo *chaves.snk* para o seu projeto no Visual Studio. Na janela **Solution Explorer**. Clique com o botão direito do mouse na raiz da aplicação. Selecione **Properties**. Em seguida, clique em **Signing**. Na caixa de combinação **Choose a strong name key file**: insira *chaves.snk*. Componentes, ou seja, DLLs assinadas com chaves criptografadas usadas pelo ASP.NET, devem ser armazenadas no Global Assembly Cache (GAC) com o auxílio da ferramenta *gacutil.exe*.

```
gacutil /i componente.dll
```

11.7 Proteção extra

Antes de momentos críticos, como alterar uma senha, acessar o cadastro de usuários, excluir produtos etc., solicite a senha novamente ao usuário. Esse é um recurso necessário, pois um usuário mal-intencionado pode reproduzir um cookie de autenticação legítimo. A solicitação da senha evita o acesso direto ao recurso protegido. E deve ser parte da estratégia de defesa em profundidade usada pela aplicação.

Outra forma de proteção extra é a utilização de CAPTCHAs. CAPTCHA é o nome atribuído às imagens com códigos numéricos, palavras ou conjunto de letras e números usados por formulários de cadastro e login. Essas imagens podem ser lidas por um ser humano, mas não por um software. Assim, evitamos que um usuário mal-intencionado utilize um processo automatizado para descobrir uma senha. A ideia central é criar um mecanismo de verificação (um porteiro) que permita somente a entrada de seres humanos e não de máquinas.

Use CAPTCHAs em todos os formulários. Atualmente, o CAPTCHA do Google é usado por milhares de websites. Para incluir o CAPTCHA do Google no seu website, acesse: <http://www.google.com/recaptcha>

e siga os passos descritos para implementação.

11.7.1 Ataques CSRF

Ataques de um único clique (CSRF) podem ser evitados com o HTML helper `AntiForgeryToken`:

```
@using (Html.BeginForm()) {  
    @Html.AntiForgeryToken()  
    @Html.ValidationSummary(true)
```

```

<fieldset>
    <legend>Nova categoria</legend>
    <div class="editor-label">
        @Html.LabelFor(model => model.CategoryName)
    </div>
    <div class="editor-field">
        @Html.TextBoxFor(model => model.CategoryName, new { @style = "width: 250px;" })
        @Html.ValidationMessageFor(model => model.CategoryName)
    </div>
    <div class="editor-label">
        @Html.LabelFor(model => model.Description)
    </div>
    <div class="editor-field">
        @Html.TextAreaFor(model => model.Description, new { @style = "width: 250px;" })
        @Html.ValidationMessageFor(model => model.Description)
    </div>
    <p>
        <input type="submit" value="Create" />
    </p>
</fieldset>
}

```

Ao executar a página o HTML helper `AntiForgeryToken` gera um valor aleatório criptografado em um campo oculto de formulário e em um cookie. O valor é semelhante a este:

```

<form action="/Home/Create" method="post">
    <input name="__RequestVerificationToken" type="hidden"
    value="3PyYVm0+Dnk02DVZzGRzKeDWTkIA/QiYgCyhjPLWZPUs/Tj8z18oH18H4QHDAacyB9csjFe8M1qVzeH5oKwZ
    zIbqw/eR84bpCQVlqfxDfUTdE1jnw6UqxP/FiGS8yAZcvU7HSqfJ+/sQewAd2PSGSfESRGsMFufvghf7J3Zzn8=" />

```

O HTML helper `AntiForgeryToken` aceita os seguintes parâmetros opcionais:

Parâmetro	Descrição
salt	Valor aleatório usado para aumentar a segurança. Pode ser algo semelhante a uma senha. Exemplo: jh45*f2d6j+djk4h6sjfg-ahsj+45fdgjd
domain	O cookie é associado a um website, e não a uma página específica.
path	Define o diretório da aplicação no qual o cookie pode ser acessado.

Obs.: quando você define o parâmetro `salt` no HTML helper `AntiForgeryToken` é necessário defini-lo também, com o mesmo valor, no atributo `ValidateAntiForgeryToken`.

O atributo `ValidateAntiForgeryToken` compara o valor do campo oculto de formulário e do cookie:

```

[ValidateAntiForgeryToken, HttpPost, Authorize(Roles = "Administradores")]
public ActionResult Create(Category entidade) {
    // Restante do código
}

```

Se for diferente, um erro é gerado. Significa que a página foi adulterada ou o mecanismo de cookie no navegador do usuário está desativado. Observe a figura 11.5:

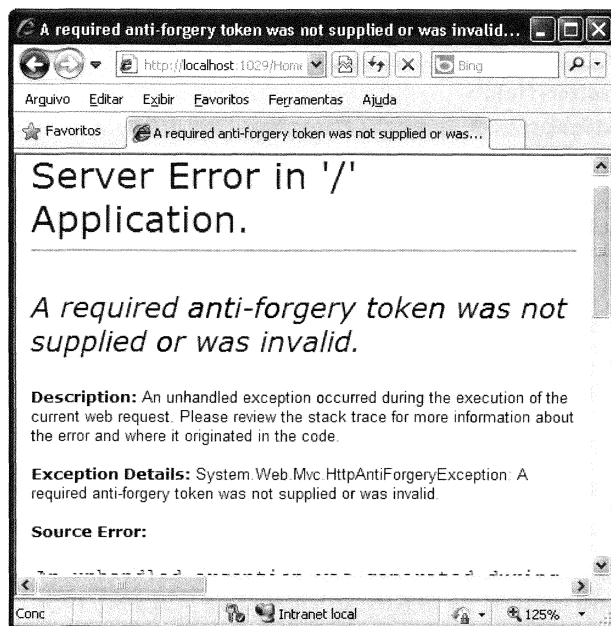


Figura 11.5 – Validação AntiForgeryToken falha.

11.7.2 Atributos ValidateInput e AllowHtml

Para aceitar elementos HTML em campos de formulário, defina o atributo `ValidateInput` como `false` em um método de controlador:

```
[HttpPost, ValidateInput(false)]
public ActionResult Create(Category entidade) {
    // Restante do código
}
```

Para permitir a entrada de elementos HTML em campos específicos, defina o atributo `AllowHtml` na propriedade correspondente ao campo:

```
[AllowHtml]
public string Description { get; set; }
```

Exemplo:

```
public class Category {
    [Display(Name = "Categoria")]
    public string CategoryName { get; set; }
```

```
[AllowHtml]
[Display(Name = "Descrição")]
public string Description { get; set; }
}
```

11.8 Validando entradas

Identifique e valide sempre todas as entradas da sua aplicação. Entenda por entrada: campos de formulários, query string, campos ocultos, estado de visualização, cookies normais e ocultos, entrada para componentes e web services.

Não confie na validação do lado do cliente (JavaScript), valide a aplicação também no lado do servidor.

Restrinja a entrada, verifique os dados válidos. Use expressões regulares:

```
[RegularExpression(@"^[a-zA-Z0-9_\/\s]{3,15}$", ErrorMessage="Digite uma categoria válida.")]
public string CategoryName { get; set; }
```

Ou verifique a entrada usando o método `TryParse` de um tipo de dados específico.

Exemplo:

```
[HttpPost]
public ActionResult Create(Product entidade) {
    Decimal resultado;
    if (!Decimal.TryParse(entidade.UnitPrice.ToString(), out resultado)) return View();
    return View();
}
```

Ao validar uma entrada verifique o tipo de dados, o comprimento, o formato e o intervalo.

11.9 Autenticação

Na autenticação o usuário se identifica para a aplicação. Os principais cuidados que devemos ter são:

- Transmitir credenciais via SSL.
- Não compartilhar o cookie de autenticação com os cookies da aplicação.
- Manter uma política rígida de senhas. As senhas devem conter letras maiúsculas, minúsculas, números e caracteres não alfanuméricos.
- Valide as entradas do formulário de login.
- Use CAPTCHAs.

11.10 Autorização

Na autorização você controla os recursos protegidos que o usuário autenticado pode acessar na aplicação.

Os principais cuidados que devemos ter na autorização de usuários são:

- Divida a aplicação em áreas públicas e restritas.
- Nunca conceda acesso irrestrito a um usuário ou grupo. Os usuários devem ter acesso limitado às funções que executam na aplicação.
- Divida os usuários em grupos de usuários.
- Use proteção múltipla, por exemplo, restrinja o acesso à área restrita no arquivo de configuração `web.config`, no método de controlador com o atributo `Authorize`, na classe de acesso de dados com segurança de acesso ao código e, no banco de dados, conceda acesso somente a stored procedures.

11.11 Dados confidenciais

Os segredos da aplicação não devem ser armazenados, se não, armazene os dados criptografados. Quanto maior for o tempo de armazenamento das informações, maior deve ser o tamanho da chave criptográfica.

A seguir, os principais algoritmos de criptografia usados pelo .NET Framework:

- Data Encryption Standard (DES) chave de 64-bit
- TripleDES chave de 128-bit ou 192-bit
- Rijndael chave de 128–256 bit
- RSA chave de 384–16,384 bit

Cuidados que devemos ter com dados sigilosos:

- Nunca armazene segredos no código ou em texto puro no banco de dados.
- Transmita as informações sigilosas via SSL ou IPsec.
- Descriptografe as informações somente quando for usá-las; em seguida, descarte-as.
- Armazene sempre hash de senhas.
- Nunca armazene informações sigilosas em cookies, query strings ou campos ocultos, cabeçalhos de resposta, cache.
- Não desenvolva seu próprio mecanismo de criptografia, use os algoritmos de criptografia do .NET Framework.

11.12 Manipulação de parâmetros

Não tome decisões importantes, como alterar preços, realizar cálculos baseando-se em campos de formulário normais ou ocultos, query string, cabeçalhos de resposta, cookies. Use-os apenas para passar identificadores de páginas.

11.13 Gerenciamento de exceções

Não revele informações que possam ser usadas por um usuário mal-intencionado. Por exemplo, não use mensagens específicas, como:

A senha está incorreta.

Use mensagens de erro genéricas:

O nome do usuário e/ou senha estão incorretos.

Proteja a aplicação, defina o atributo `mode` do elemento como `customErrors`:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.web>
    <customErrors mode="RemoteOnly"/>
  </system.web>
</configuration>
```

11.14 Log de erros

O objetivo do log é identificar as atividades e anomalias da aplicação. Crie um arquivo de log com todas as exceções ocorridas na aplicação e o armazene no banco de dados da aplicação ou use um banco de dados separado. Um ótimo local para gravar o log é no arquivo `Global.asax`:

```
void Application_Error(object sender, EventArgs e) {
    Exception ex = Server.GetLastError();
    // ex.Message
    Server.ClearError();
}
```

No ASP.NET MVC temos a possibilidade de criar um filtro ação global para gravar as informações no log.

Armazene a mensagem retornada pela propriedade `Message` em um banco de dados ou no log de eventos do Windows. Além das exceções, você pode logar as páginas restritas acessadas pelos usuários do website e toda a atividade do banco de dados.

Obs.: não efetue o log de informações sigilosas.

11.15 Aplicando SSL

Os diretórios que contêm os arquivos que acessam recursos restritos ou de autenticação de usuários devem ser protegidos com SSL.

Aplique o atributo `RequireHttps` aos métodos de controlador que exigem SSL. Como o método `Create`:

```
[HttpPost, RequireHttps, Authorize(Roles = "Administradores")]
public ActionResult Create(Category entidade) {
    return View();
}
```

E `Edit`:

```
[HttpPost, RequireHttps, Authorize(Roles = "Administradores")]
public ActionResult Edit(Category entidade) {
    return View();
}
```

E os métodos de controlador de autenticação de usuários, como `LogOn`:

```
[HttpPost, RequireHttps]
public ActionResult LogOn(LogOnModel model, string returnUrl) {
    // Restante do código
}
```

O método `Register`:

```
[HttpPost, RequireHttps]
public ActionResult Register(RegisterModel model) {
    // Restante do código
}
```

E `ChangePassword`:

```
[Authorize, HttpPost, RequireHttps]
public ActionResult ChangePassword(ChangePasswordModel model) {
    // Restante do código
}
```

11.15.1 Configurando SSL no Servidor web

Para criar um certificado de teste use a ferramenta `makecert` do .NET Framework:

```
makecert -r -pe -n "CN=NomeDoComputador" -b 01/01/2000 -e 01/01/2036 -eku 1.3.6.1.5.5.7.3.1 -ss
my -sr localMachine -sky exchange -sp "Microsoft RSA SChannel Cryptographic Provider" -sy 12
```

Clique no menu **Iniciar > Todos os programas > Microsoft Visual Studio 2013 > Visual Studio Tools > Developer Command Prompt for VS2013**. Em seguida, digite:

```
makecert -r -pe -n "CN=alfredo" -b 01/01/2000 -e 01/01/2036 -eku 1.3.6.1.5.5.7.3.1 -ss my -sr  
localMachine -sky exchange -sp "Microsoft RSA SChannel Cryptographic Provider" -sy 12
```

O nome do computador é obrigatório e é usado para acessar a página via SSL.

Exemplo:

`https://alfredo/capitulo11/home`

No IIS 7 você tem a opção de usar localhost na URL em vez do nome do servidor:

`https://localhost/capitulo11/home`

Após a criação do certificado de teste com a ferramenta makecert, configure o IIS. Siga os passos descritos a seguir:

- No menu **Iniciar** do Windows, clique em **Executar....**
- Na caixa **Abrir**, digite `inetmgr` e, em seguida, clique em **OK**.
- Em seguida, clique com o botão direito do mouse em **Site da Web padrão** e selecione **Propriedades**. Observe a figura 11.6:

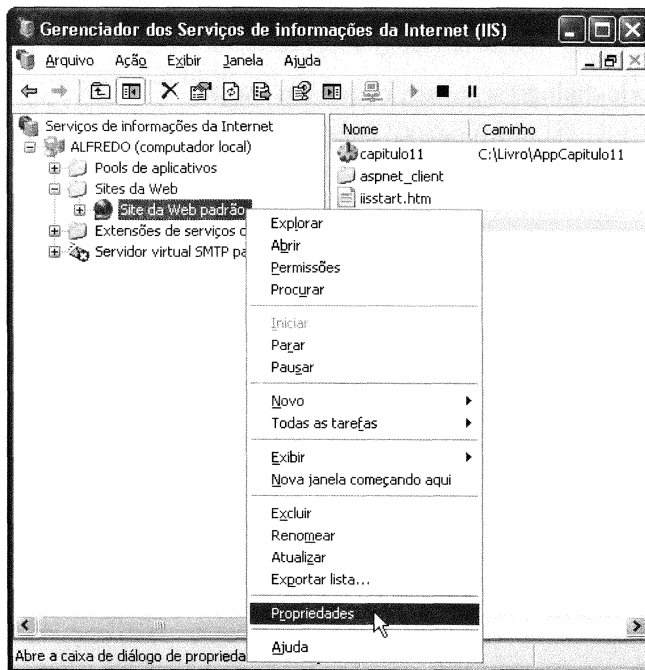


Figura 11.6 – Gerenciador do IIS.

- Na caixa de diálogo **Propriedades de Site da Web padrão**, clique na aba **Segurança de diretório** e, em seguida, em **Certificado de servidor....** Observe a figura 11.7:

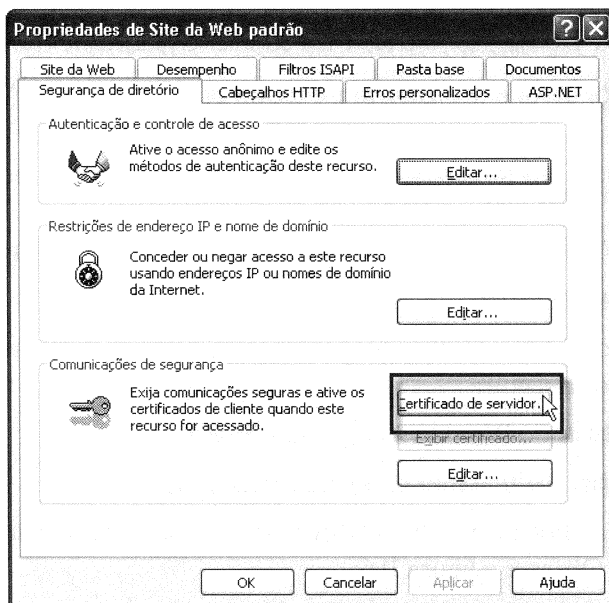


Figura 11.7 - Caixa de diálogo Propriedades de Site da Web padrão.

- Na caixa de diálogo seguinte, clique em **Avançar**.
- Na caixa de diálogo **Assistente de certificado do IIS**, marque a caixa de opção **Atribuir um certificado existente**. Clique em **Avançar**, conforme a figura 11.8:

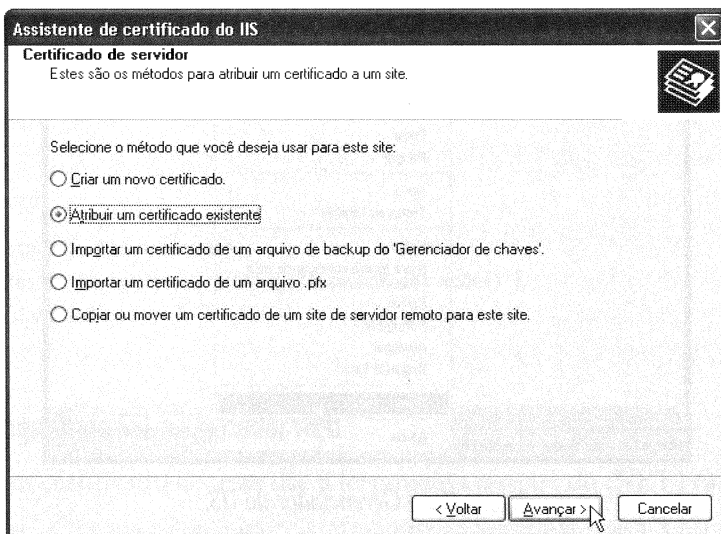


Figura 11.8 – Caixa de diálogo Certificado de servidor.

- Na caixa de diálogo **Certificados disponíveis**, selecione o certificado criado para o nome do seu computador. Observe a figura 11.9:

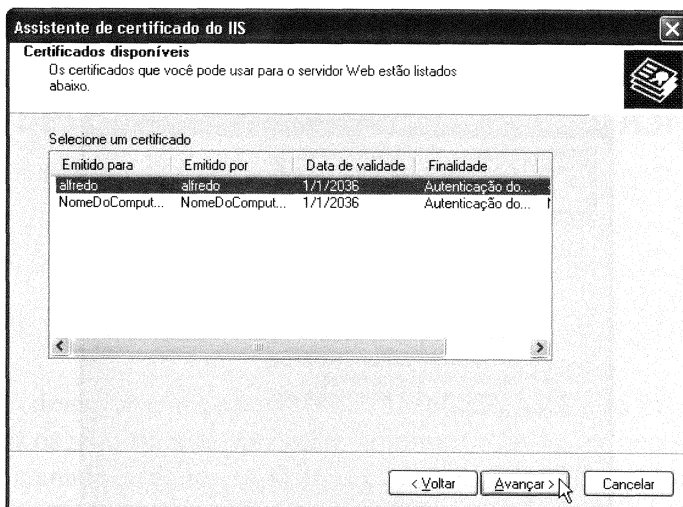


Figura 11.9 – Lista de certificados disponíveis.

Clique em **Avançar** nas telas seguintes e em **Concluir**.

11.15.2 Aplicando SSL a um diretório

Aplique SSL aos diretórios Admin e Account. No gerenciador do IIS, clique com o botão direito do mouse no diretório Admin e, em seguida, em **Propriedades**. Na aba **Segurança de diretório**, clique em **Editar**. Observe a figura 11.10:

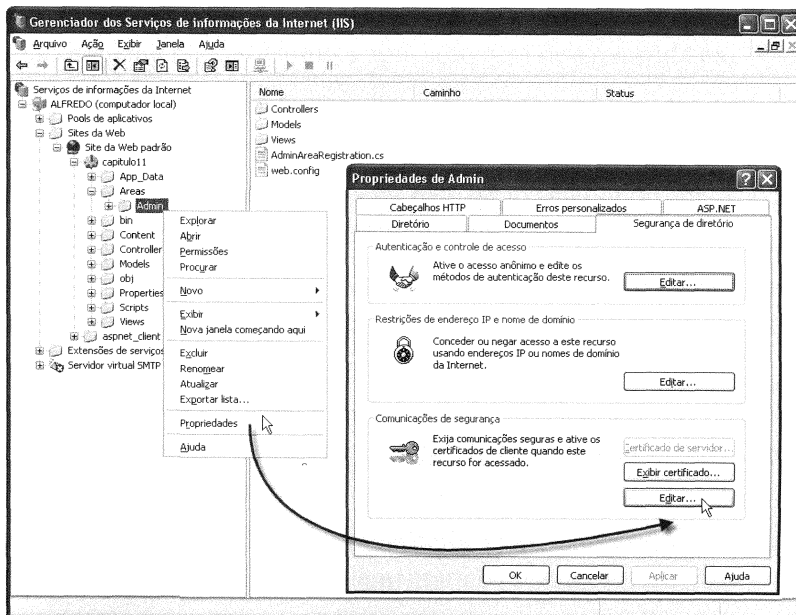


Figura 11.10 – Selecionando o diretório Admin para aplicar SSL.

Na caixa de diálogo **Comunicações de segurança**, marque a caixa de opção **Exigir canal de segurança (SSL)**, conforme a figura 11.11:

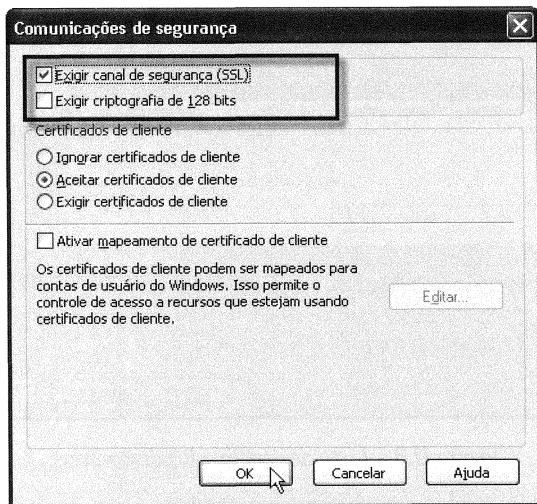


Figura 11.11 – Aplicando SSL ao diretório Admin.

Repita os passos anteriores para o diretório Account e, em seguida, acesse as páginas protegidas com SSL, prefixo https, normalmente:

<https://localhost/capitulo11/Account/LogOn>

Desenvolvendo um website com o ASP.NET MVC

Neste capítulo desenvolvemos passo a passo um pequeno website. Para cada recurso determinamos os requisitos da aplicação, definimos o layout, se necessário criamos as tabelas e a camada de dados, validamos as entradas e codificamos os métodos de controlador.

Não há necessidade de você digitar todas as marcações HTML das páginas. Baixe os arquivos de exemplo no website da Editora Novatec e monte as páginas. Se você tiver dúvidas ou sugestões sobre a aplicação, entre em contato comigo. E-mail de contato: alfredolotar@gmail.com

12.1 Especificações

A elaboração das especificações é o primeiro passo no desenvolvimento de qualquer aplicação. É nesse estágio que você define o que é e o que faz a aplicação. Quais recursos serão implementados. Como a aplicação será desenvolvida etc.

Podemos dividir as especificações em diversos níveis ou tipos. As especificações de nível 1, digamos assim, são elaboradas em conjunto com o cliente. ✓

As especificações de nível 2 são elaboradas pelo programador ou analista de sistemas. São mais completas e podem ser subdivididas em diversos níveis. ✓

Vamos supor que um cliente solicita uma aplicação de comércio eletrônico para um programador e informa que deseja que o website tenha um mecanismo de busca e que aceite duas formas de pagamento – boleto e cartão de crédito. ✓

A partir dessas informações, o programador pode elaborar as especificações de nível 2, ou seja, você lista todos os recursos essenciais, importantes e acessórios em um website de comércio eletrônico. ✓

Após a pesquisa, análise e definição das especificações. Chegamos à conclusão de que o website de exemplo deste livro implementará os seguintes recursos:

- Página inicial.
- Mecanismo de busca.

- Menu exibindo as categorias de produtos.
- Lista de produtos por categoria.
- Carrinho de compras.
- Cadastro de clientes.
- Login de clientes.

Como já mencionamos, as especificações podem ser divididas em subníveis, por exemplo: podemos pegar o recurso mecanismo de busca e subdividi-lo, ou seja, você precisa ser cada vez mais objetivo e específico.

Elaborar especificações significa tomar uma série de decisões sobre o que, quando e como usar cada recurso, tecnologia, função, classe, imagem, banco de dados, texto etc.

Ao subdividir a especificação mecanismo de busca pense:

- Quais informações devem ser exibidas no mecanismo de busca.
- Como funciona o mecanismo de busca.
- Quando essas informações serão exibidas.
- Como será o design:
 - Qual a cor e tipo de fonte.
 - Como será o layout.
 - Quais imagens precisam ser criadas.
- Quais recursos serão utilizados. Exemplo: cookies, variáveis de sessão, cache, registro etc.
- Que tabelas do banco de dados serão utilizadas.
- Quais instruções LINQ, SQL ou stored procedures serão utilizadas.
- Que funções, classes, métodos, propriedades, eventos serão criados.
- Quais controles do ASP.NET ou HTML helpers serão utilizados.

Após a definição das especificações – recursos que o nosso website de comércio eletrônico deve implementar –, definimos quais informações serão exibidas.

Ao desenvolver um website ou recurso específico da aplicação, é importante seguir os seguintes passos:

- Determine os requisitos do recurso ou aplicação.
- Defina quais informações cada recurso deve conter.
- Projete o design da aplicação ou recurso.
- No banco de dados defina as tabelas e os campos necessários.
- Acrescente a tabela ao modelo conceitual do ADO.NET Entity Framework.
- Desenvolva a camada de dados.

- Valide as propriedades referentes a cada tabela.
- Crie os métodos de controlador.
- Desenvolva os views, ou seja, a camada de apresentação.
- Monte e teste a aplicação.

12.1.1 Quais recursos devo usar

Ao projetar um recurso específico do website tenha em mente as seguintes questões:

- O recurso deve ser colocado no cache? Sim ou não?
- O modo de programação Ajax se aplica? Sim ou não?
- O recurso usa cookies? Sim ou não?
- O recurso usa variáveis de sessão? Sim ou não?
- O recurso necessita de paginação? Sim ou não?
- Há necessidade de validação de entradas e de propriedades? Sim ou não?
- É um recurso restrito?
- SSL é necessário? Sim ou não?
- É necessário aplicar atributos de segurança (`RequireHttps`, `Authorize`, `ValidateAntiForgeryToken`, `Bind`)? Sim ou não?
- É necessário registrar o acesso ao recurso no log? Sim ou não?
- É necessário criptografar as informações no banco de dados? Sim ou não?
- Deve ser aplicada a segurança de acesso ao código? Sim ou não?

A seguir listamos as informações contidas em cada recurso do website, ou seja, definimos as especificações. A figura 12.1 ilustra a relação entre os recursos da aplicação de exemplo:

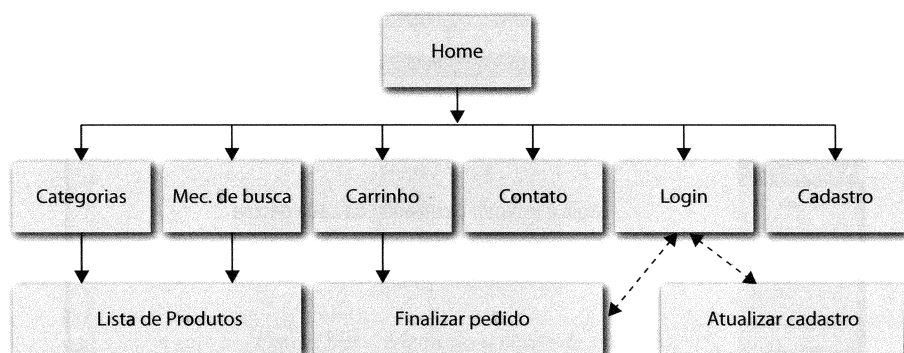


Figura 12.1 – Recursos do website de exemplo.

Obs.: sugerimos que você implemente o recurso finalizar pedido e atualizar cadastro.

11.2 Criando uma nova aplicação

Para desenvolver o website de exemplo abordado neste capítulo, inicie o Visual Studio. Acesse o menu **File** e clique em **New Project**

Na caixa de diálogo **New Project** selecione **Web > Visual Studio 2012 > ASP.NET MVC 4 Web Application**. Nomeie o projeto como **AppCapítulo12** e clique em **OK**.

Em seguida, na caixa de combinação **View Engine**, selecione o mecanismo de exibição **Razor**, selecione o template **Internet Application**. Observe a figura 11.1.

12.3 Telas da aplicação

Após elaborar as especificações, você, como projetista da aplicação, precisa decidir quais informações cada página deve conter.

Com base nessas informações você tem condições de elaborar o design da página, criar as tabelas com seus respectivos campos. E, é claro, iniciar a codificação e os testes.

Nas páginas a seguir observe as figuras e veja as informações contidas em cada página da aplicação de exemplo, como a página de layout do website que contém o conteúdo comum a todas as páginas. A figura 12.2 ilustra o conteúdo da página de layout.



Figura 12.2 – Conteúdo comum a todas as páginas.

A página inicial do website exibe o produto mais recente de cada categoria, conforme a figura 12.3.

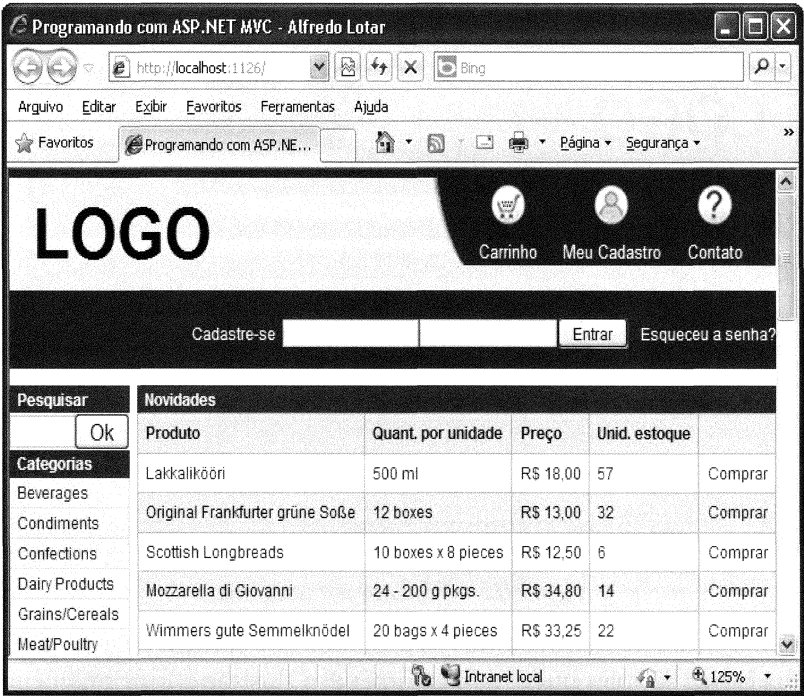


Figura 12.3 – Página inicial do website.

O menu contém todas as categorias cadastradas na tabela Categories do banco de dados Northwind (Figura 12.4).

Categorias
Beverages
Condiments
Confections
Dairy Products
Grains/Cereals
Meat/Poultry
Produce
Seafood
Computadores

Figura 12.4 – Menu do website de exemplo.

Ao clicar em determinada categoria o usuário exibe os produtos da categoria. Se necessário, o mecanismo de paginação é acionado. Observe a figura 12.5.



Figura 12.5 – Produtos por categoria.

O mecanismo de busca retorna todos os produtos que contêm o termo de busca, ou seja, é uma busca simples que visa basicamente a ilustrar o funcionamento de um sistema de busca. Observe a figura 12.6.

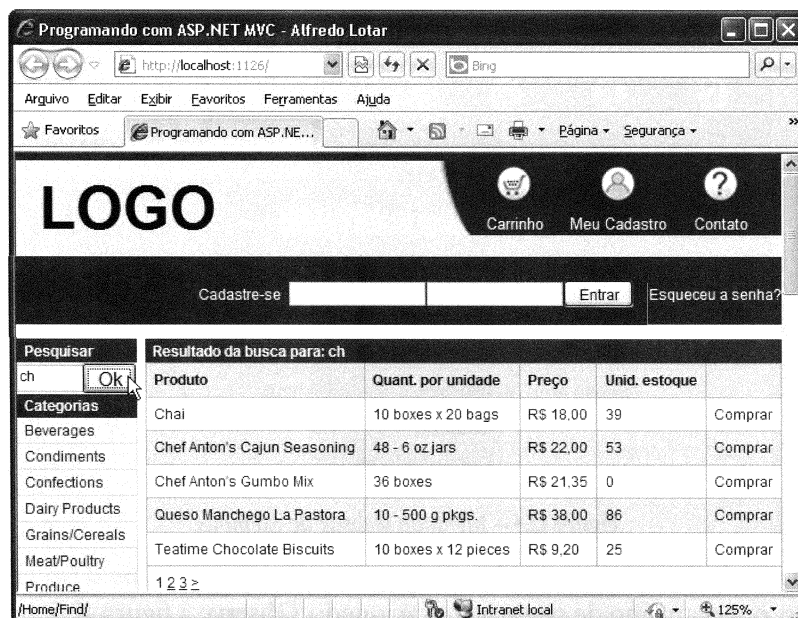


Figura 12.6 – Resultado da busca com paginação.

O carrinho de compras contém os itens selecionados pelo usuário/cliente. Ao acessar o carrinho de compras podemos visualizar e excluir os itens. Observe a figura 12.7.

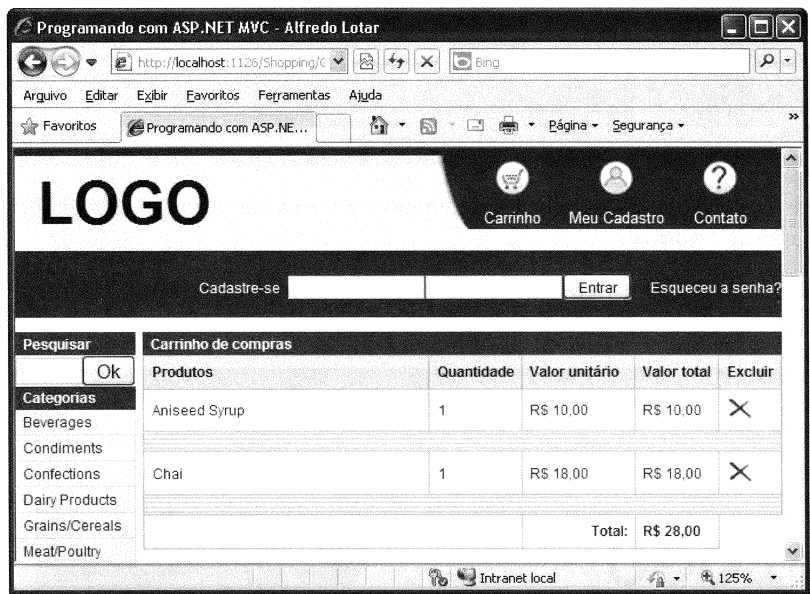


Figura 12.7 – Carrinho de compras.

O formulário de contato envia as informações digitadas para uma tabela do banco de dados. Veja a figura 12.8.

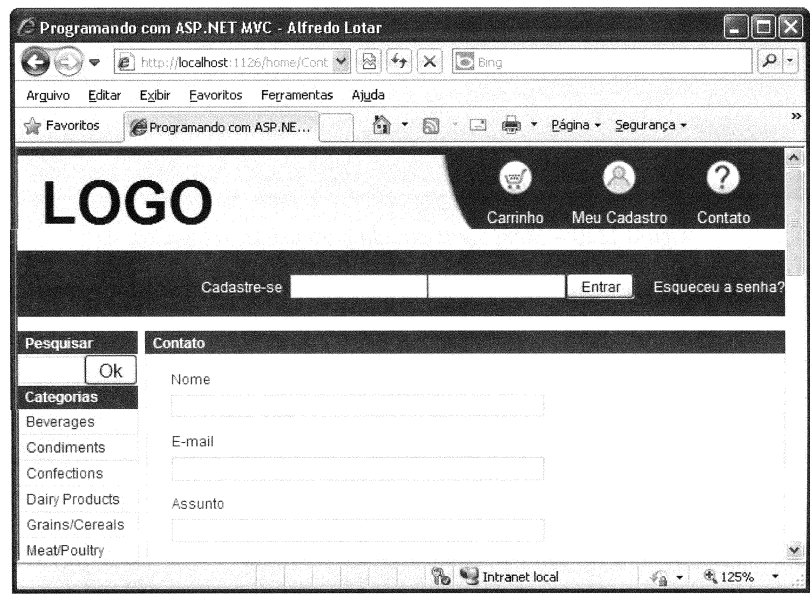


Figura 12.8 – Formulário de contato.

A tela de login exibe um formulário simples com dois campos para usuários anônimos. Observe a figura 12.9.

Figura 12.9 – Formulário de login.

Obs.: o formulário de login aparece meio espremido na figura, mas na aplicação a aparência é normal.

Para os usuários logados ocultamos o formulário de login e exibimos a data do último login e o nome de usuário. Veja a figura 12.10.

Produto	Quant. por unidade	Preço	Unid. estoque	
Chai	10 boxes x 20 bags	R\$ 18,00	39	Comprar
Chef Anton's Cajun Seasoning	48 - 6 oz jars	R\$ 22,00	53	Comprar
Chef Anton's Gumbo Mix	36 boxes	R\$ 21,35	0	Comprar
Queso Manchego La Pastora	10 - 500 g pkgs.	R\$ 38,00	86	Comprar
Teatime Chocolate Biscuits	10 boxes x 12 pieces	R\$ 9,20	25	Comprar

Figura 12.10 – Mensagem exibida para usuários logados.

O cadastro de usuários é realizado em duas etapas. A primeira solicita as informações de login. Observe a figura 12.11.

The screenshot shows a web browser window titled "Programando com ASP.NET MVC - Alfredo Lotar". The address bar shows "http://localhost:1126/Account/re". The browser has a menu bar with "Arquivo", "Editar", "Exibir", "Favoritos", "Ferramentas", and "Ajuda". Below the menu bar is a "Favoritos" section with a link to "Programando com ASP.NET...". The main content area has a dark header with "Cadastre-se" and "Entrar" buttons, and a link "Esqueceu a senha?". Below this is a "Pesquisar" section with an "Ok" button and a "Categorias" list: Beverages, Condiments, Confections, Dairy Products, Grains/Cereals, Meat/Poultry, Produce, Seafood, and Computadores. The main form is titled "Crie uma nova conta" and contains fields for "Usuário", "E-mail", "Senha", and "Confirme a senha", followed by a "Registrar" button. The status bar at the bottom shows "Intranet local" and "125%".

Figura 12.11 – Formulário de cadastro de login.

E a segunda etapa solicita ao usuário as informações pessoais, conforme a figura 12.12.

The screenshot shows a web browser window titled "Programando com ASP.NET MVC - Alfredo Lotar". The address bar shows "http://localhost:1126/Account/Cr". The browser has a menu bar with "Arquivo", "Editar", "Exibir", "Favoritos", "Ferramentas", and "Ajuda". Below the menu bar is a "Favoritos" section with a link to "Programando com ASP.NET...". The main content area has a dark header with "LOGO" and navigation links: "Carrinho", "Meu Cadastro", and "Contato". Below this is a status bar with "Último login: 1/9/2011 19:17:30", "Seja bem vindo novatec", and "Sair". The main form is titled "Cadastro de usuários - continuação ..." and contains fields for "Nome", "Cpf", and "Endereço". The "Cpf" field has a value of "12" and a message "Cpf inválido. Ex.: 00.00.00.000-00". The status bar at the bottom shows "Concluído" and "Intranet local".

Figura 12.12 – Formulário de cadastro de informações pessoais.

Para quem esqueceu a senha temos o formulário que envia para o e-mail cadastrado uma nova senha. Observe a figura 12.13.



Figura 12.13 – Formulário de envio de senhas.

12.4 Layout

O layout dos websites difere pouco um do outro. Isso é bom porque deixa o usuário confortável e facilita a navegação.

Geralmente, um website típico tem um logotipo (no topo) e um menu, ambos à esquerda. A parte superior do website, ao lado do logotipo, é usada para exibir anúncios e chamar a atenção para produtos e serviços oferecidos pelo website; já a parte inferior contém informações sobre contato e copyright, entre outras informações. O conteúdo do website é exibido no centro, podendo ou não conter outras subseções.

Na figura 12.14 listamos os principais tipos de layouts.

O layout varia de acordo com o tipo de website e o conteúdo exibido pelas páginas. Por exemplo, um website de uma imobiliária exibe pouco texto e fotos grandes dos imóveis, enquanto um website de notícias exibe pequenos blocos de texto e algumas fotos pequenas.

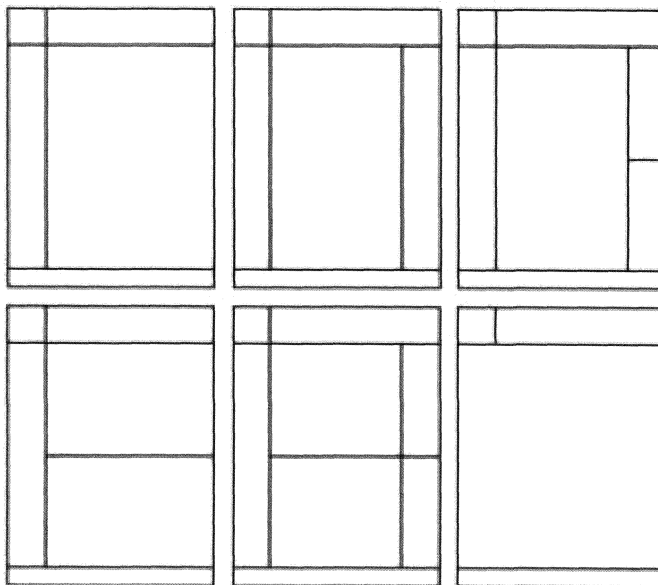


Figura 12.14 – Tipos de layouts de páginas de websites.

O conteúdo comum a todas as páginas deve ser formatado com seletores CSS e colocado em uma master page – página de layout como é denominada quando usamos a sintaxe Razor.

O layout de um website é dividido em pequenas tabelas. Cada tabela delimita uma área da página.

Para formatar as tabelas da aplicação substituímos no arquivo Content\Site.css:

```
body {
    background-color: #5c87b2;
    font-size: 75%;
    font-family: Verdana, Tahoma, Arial, "Helvetica Neue" , Helvetica, Sans-Serif;
    margin: 0;
    padding: 0;
    color: #696969;
}
```

Por:

```
body {
    font-size: 75%;
    font-family: Verdana, Tahoma, Arial, "Helvetica Neue" , Helvetica, Sans-Serif; color:
    #232323; background-color: #fff; margin-left: 0; margin-right: 0; margin-top: 0; margin-
    bottom: 0;
}
```

Substituímos também o seletor Table:

```
Table {
    border: solid 1px #e8eef4;
    border-collapse: collapse;
}

table td {
    padding: 5px;
    border: solid 1px #e8eef4;
}

table th {
    padding: 6px 5px;
    text-align: left;
    background-color: #e8eef4;
    border: solid 1px #e8eef4;
}
```

Por:

```
table {
    border-collapse: collapse;
    border: 0;
}

td {
    padding: 0;
}
```

A aplicação de exemplo deste livro contém as informações comuns a todas as páginas no arquivo Views\Shared_Layout.cshtml. Este arquivo é adicionado quando criamos um novo projeto ASP.NET MVC Razor e contém o seguinte código:

```
<!DOCTYPE html>
<html>
    <head>
        <title>@ViewBag.Title</title>
        <link href="@Url.Content("~/Content/Site.css")" rel="stylesheet" type="text/css" />
        <script src="@Url.Content("~/Scripts/jquery-1.4.4.min.js")" type="text/javascript"></script>
    </head>
    <body>
        <div class="page">
            <div id="header">
                <div id="title">
                    <h1>My MVC Application</h1>
                </div>
```

```

        <div id="logindisplay">
            @Html.Partial("_LogOnPartial")
        </div>

        <div id="menucontainer">

            <ul id="menu">
                <li>@Html.ActionLink("Home", "Index", "Home")</li>
                <li>@Html.ActionLink("About", "About", "Home")</li>
            </ul>

        </div>
    </div>

    <div id="main">
        @RenderBody()
    <div id="footer">
    </div>
    </div>
</div>
</body>
</html>

```

Altere o arquivo de layout `Views\Shared\_Layout.cshtml` conforme instruções a seguir. Exclua todo o conteúdo dentro da tag `body`:

```

<div class="page">
    <div id="header">
        <div id="title">
            <h1>My MVC Application</h1>
        </div>

        <div id="logindisplay">
            @Html.Partial("_LogOnPartial")
        </div>

        <div id="menucontainer">
            <ul id="menu">
                <li>@Html.ActionLink("Home", "Index", "Home")</li>
                <li>@Html.ActionLink("About", "About", "Home")</li>
            </ul>
        </div>
    </div>

```

```

<div id="main">
    @RenderBody()
    <div id="footer">
    </div>
</div>
</div>

```

A tabela a seguir insere a estrutura do topo da página de layout:

```

<table style="text-align: center; border: 0; width: 100%">
    <tr>
        <td style="background-color: #006486; height: 5px;" colspan="4">
        </td>
    </tr>
    <tr>
        <td style="background-color: #edf2f2; height: 65px; width: 15px;">
        </td>
        <td style="background-color: #edf2f2;">
            <!-- Logo aqui -->
        </td>
        <td style="background-color: #edf2f2; text-align: right; vertical-align: top; width: 25%;">
            <!-- Menu superior aqui -->
        </td>
    </tr>
</table>

```

Após remover o código gerado para a página de layout, insira a tabela anterior na página de layout `Views\Shared\_Layout.cshtml`, conforme instruções no código a seguir:

```

<!DOCTYPE html>
<html>
    <head>
        <title>@ViewBag.Title</title>
        <link href="@Url.Content("~/Content/Site.css")" rel="stylesheet" type="text/css" />
        <script src="@Url.Content("~/Scripts/jquery-1.4.4.min.js")" type="text/javascript"></script>
    </head>

    <body>
        <!--Código da tabela anterior aqui-->
    </body>
</html>

```

Observe a figura 12.15.

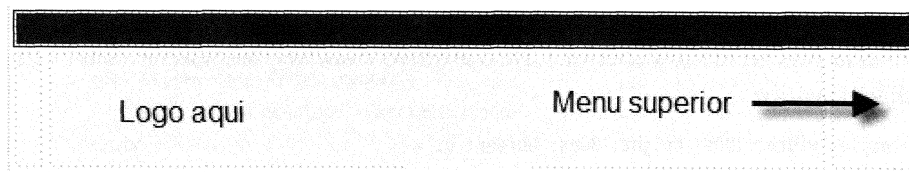


Figura 12.15 – Estrutura do topo do website.

Em seguida, inserimos a tabela com o logo diretamente na marca “Logo aqui”:

```
<table style="width: 100%; border: 0px;">
  <tr>
    <td style="width: 100%; text-align: justify; color:black; font-size: 40px; font-family: Arial;">
      <strong>LOGO</strong>
    </td>
  </tr>
</table>
```

Exemplo:

```
<table style="text-align: center; border: 0; width: 100%">
  <tr>
    <td style="background-color: #006486; height: 5px;" colspan="4">
    </td>
  </tr>
  <tr>
    <td style="background-color: #edf2f2; height: 65px; width: 15px;">
    </td>
    <td style="background-color: #edf2f2;">
      <table style="width: 100%; border: 0px;">
        <tr>
          <td style="width: 100%; text-align: justify; color:black; font-size: 40px;
            font-family: Arial;">
            <strong>LOGO</strong>
          </td>
        </tr>
      </table>
    </td>
    <td style="background-color: #edf2f2; text-align: right; vertical-align: top; width:
      25%;">
      <!-- Menu superior aqui -->
    </td>
  </tr>
</table>
```

Por meio do partial view Views/Home/MenuSuperior.cshtml:

```
@{Html.RenderPartial("~/Views/Home/MenuSuperior.cshtml");}
```

Inserimos na marca “Menu superior aqui” a tabela que contém o texto, o link e as figuras relativas ao menu superior. Crie o arquivo Views/Home/MenuSuperior.cshtml e insira o código a seguir:

```
<table style="width: 100%; height: 30px; border: 0;">
  <tr>
    <td rowspan="2" style="vertical-align: top; text-align: right; background-color: #edf2f2">
      
    </td>
    <td style="width: 25%; background-color: #006486">
      <div style="text-align: center">
        <a href="@Href("~/shopping/GetShoppingCartItems")">
          
        </a>
      </div>
    </td>
    <td style="width: 35%; background-color: #006486">
      <div style="text-align: center">
        <a href="@Href("~/account/AtualizarCadastro")">
          
        </a>
      </div>
    </td>
    <td style="width: 25%; background-color: #006486">
      <div style="text-align: center">
        <a href="@Href("~/home/Contato")">
          
        </a>
      </div>
    </td>
    <td style="background-color: #006486; width: 80px;">
    </td>
    <td style="background-color: #006486; width: 50px;">
    </td>
  </tr>
```

```

<tr>
  <td style="background-color: #006486;">
    <div style="text-align: center">
      <span class="menutopo">Carrinho</span>
    </div>
  </td>
  <td style="background-color: #006486;">
    <div style="text-align: center">
      <span class="menutopo">Meu Cadastro</span>
    </div>
  </td>
  <td style="background-color: #006486;">
    <div style="text-align: center">
      <span class="menutopo">Contato</span>
    </div>
  </td>
  <td style="background-color: #006486;">
  </td>
  <td style="background-color: #006486;">
  </td>
</tr>
</table>

```

Obs.: as figuras Content/Imagens/esq_aba.gif, Content/Imagens/carrinho.gif, Content/Imagens/cadastro.gif e Content/Imagens/top_duvidas.gif definem, respectivamente, o canto esquerdo da figura, logo do carrinho de compras, do cadastro e de contato. Faça o download dos arquivos de exemplo do livro e copie as figuras para o seu projeto.

O seletor CSS `menutopo` define a formatação do texto “Carrinho”, “Meu Cadastro”, “Contato” e deve ser inserido no arquivo `Content/Site.css`:

```

.menutopo {
  color: #ffffff;
  font-size: 11px;
  font-family: Arial, Helvetica, sans-serif;
}

```

Observe a figura 12.16.

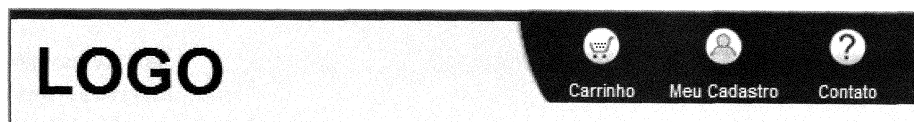


Figura 12.16 – Logo e menu superior.

Em seguida, após a última tag `</table>` dentro da página de layout `Views\Shared\_Layout.cshtml`, acrescentamos a tabela referente ao formulário de login e as mensagens de erro:

```
<table style="width: 100%; text-align: center; border: 0;">
    <tr>
        <td style="background-color: red; display: none; height: 25px;">
        </td>
    </tr>
    <tr>
        <td style="height: 45px; background-color: #006486">
            <!-- Formulário de login aqui -->
        </td>
    </tr>
</table>
```

Obs.: quando não há um local predefinido para a tabela, insira-a sempre abaixo das demais.

Novamente, após a última tag `</table>` insira a tabela referente ao conteúdo da página, o mecanismo de busca e o menu:

```
<table style="width: 100%; text-align: center; border: 0;">
    <tr>
        <td colspan="3" style="height: 10px;">
        </td>
    </tr>
    <tr>
        <td style="vertical-align: top; width: 15%">
            <!-- Mecanismo de busca e menu aqui -->
        </td>
        <td style="vertical-align: top; width: 1%">
        </td>
        <td style="vertical-align: top; height: 800px; width: 80%">
            @RenderBody()
        </td>
    </tr>
</table>
```

`RenderBody` exibe o conteúdo do view ao qual o layout está associado. E `@ViewBag.Categoria` apresenta na tela o título da seção.

Observe a figura 12.17.

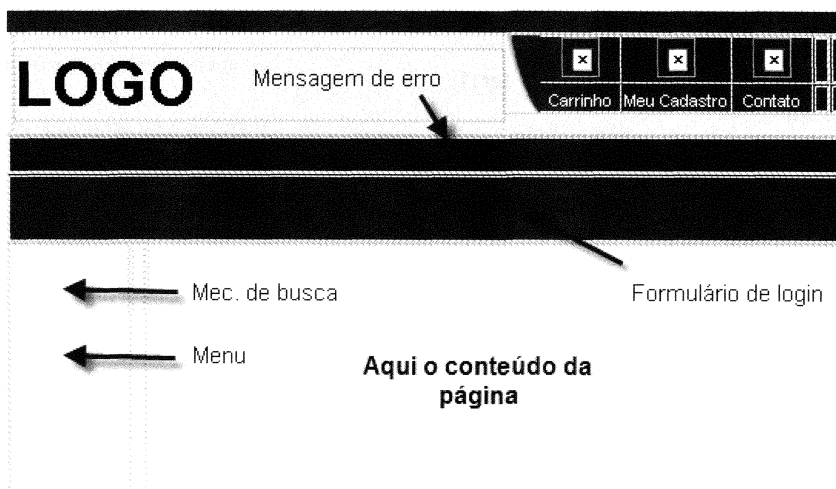


Figura 12.17 – Determina as áreas específicas da página.

O rodapé é concebido com o auxílio da tag HTML `div` e inserido ao final da página de layout `Views\Shared\_Layout.cshtml`:

```
<div style="text-align: center; vertical-align: bottom">
  <a class="navegacao" href="/">Home</a> | <a class="navegacao" href="NaoAtivada.htm">
  Como comprar</a> | <a class="navegacao" href="NaoAtivada.htm">formas de pagamento</a>
  | <a class="navegacao" href="NaoAtivada.htm">Segurança</a> | <a class="navegacao"
  href="NaoAtivada.htm">Trocas e devoluções</a> | <a class="navegacao" href="NaoAtivada.htm">
  Dúvidas (FAQ)</a> | <a class="navegacao" href="NaoAtivada.htm">Tabela de fretes</a><br />
  <a class="navegacao" href="NaoAtivada.htm">Política de privacidade</a> |
  <a class="navegacao" href="Contato.aspx">Fale conosco</a> |
  <a class="navegacao" href="NaoAtivada.htm">Quem
  somos</a>
</div>
```

E é formatado pelo seletor CSS `navegacao`:

```
.navegacao:link {
  text-decoration: none;
  font-family: Arial, Helvetica, sans-serif;
  color: blue;
  font-size: 10px;
}
.navegacao:visited {
  text-decoration: none;
  font-family: Arial, Helvetica, sans-serif;
  color: blue;
  font-size: 10px;
}
```

```
.navegacao:active {
    text-decoration: none;
    font-family: Arial, Helvetica, sans-serif;
    color: blue;
    font-size: 10px;
}
.navegacao:hover {
    text-decoration: none;
    font-family: Arial, Helvetica, sans-serif;
    color: #0099FF;
    font-size: 10px;
}
```

12.5 Desenvolvendo a camada de dados

A aplicação de exemplo deste capítulo usa um repositório genérico, ou seja, desenvolvemos uma interface com a assinatura dos métodos e, em seguida, criamos a classe que implementa esses métodos.

O código da interface `IGenericRepository` é inserido no arquivo `IGenericRepository.cs`:

```
using System;
using System.Collections.Generic;
using System.Data.Objects;
using System.Data.Objects.DataClasses;
using System.Linq.Expressions;

namespace AppCapitulo12.Models {
    public interface IGenericRepository : IDisposable {
        IEnumerable<T> GetAll<T>();
        IEnumerable<T> Find<T>(Expression<Func<T, bool>> predicate);
        T GetSingle<T>(Expression<Func<T, bool>> predicate);
        ObjectResult<T> ExecuteSQL<T>(string sql, params object[] parameters);
        void Create<T>(T entidade);
        void Edit<T>(T entidade, Expression<Func<T, bool>> predicate) where T : class, IEntityWithKey;
        void Delete<T>(Expression<Func<T, bool>> predicate);
    }
}
```

E o código da classe `EntityFrameworkRepository` que implementa os métodos da interface `IGenericRepository` é inserido no arquivo `EntityFrameworkRepository.cs`:

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Data.Objects;
using System.Data.Objects.DataClasses;
```

```

using System.Linq;
using System.Linq.Expressions;

namespace AppCapitulo12.Models {
    public class EntityFrameworkRepository : IGenericRepository {
        private ObjectContext _contexto;
        private bool disposed = false;

        public EntityFrameworkRepository(ObjectContext dataContext) {
            _contexto = dataContext;
        }

        public IEnumerable<T> GetAll<T>() {
            string entityName = GetEntidade<T>();
            IList<T> query = _contexto.CreateQuery<T>(entityName).ToList();
            return query;
        }

        public IEnumerable<T> Find<T>(Expression<Func<T, bool>> predicate) {
            string entityName = GetEntidade<T>();
            var model = _contexto.CreateQuery<T>(entityName).Where(predicate).ToList();
            return model;
        }

        public T GetSingle<T>(Expression<Func<T, bool>> predicate) {
            string entityName = GetEntidade<T>();
            var model = _contexto.CreateQuery<T>(entityName).Where(predicate).FirstOrDefault<T>();
            return model;
        }

        public ObjectResult<T> ExecuteSQL<T>(string sql, params object[] parameters) {
            return _contexto.ExecuteStoreQuery<T>(sql, parameters);
        }

        public void Create<T>(T entidade) {
            if (entidade != null) {
                string entityName = GetEntidade<T>();
                _contexto.AddObject(entityName, entidade);
                _contexto.SaveChanges();
            }
        }

        public void Edit<T>(T entidade, Expression<Func<T, bool>> predicate) where T : class,
            IEntityWithKey {
            if (entidade != null) {
                string entityName = GetEntidade<T>();

```

```

        var model = GetSingle(predicate);
        if (model != null) {
            _contexto.ApplyCurrentValues(model.EntityKey.EntitySetName, entidade);
            _contexto.SaveChanges();
        }
    }
}

public void Delete<T>(Expression<Func<T, bool>> predicate) {
    var model = GetSingle(predicate);
    if (model != null) {
        _contexto.DeleteObject(model);
        _contexto.SaveChanges();
    }
}

private string GetEntidade<T>() {
    var EntitySetName = _contexto.GetType().GetProperties()
        .Single(p => p.PropertyType.IsGenericType && typeof(IEnumerable<>)
            .MakeGenericType(typeof(T)).IsAssignableFrom(p.PropertyType));
    return EntitySetName.Name;
}

public void Dispose() {
    Dispose(true);
    GC.SuppressFinalize(this);
}

~EntityFrameworkRepository() {
    Dispose(false);
}

protected virtual void Dispose(bool disposing) {
    if (!this.disposed) {
        if (disposing) {
            if (_contexto != null) {
                _contexto.Dispose();
                _contexto = null;
            }
        }
    }
    disposed = true;
}
}
}

```

Em seguida, acrescentamos a aplicação às classes do ADO.NET Entity Framework. Para não tornar o livro repetitivo não reproduzimos aqui os passos já descritos no tópico “4.3 ADO.NET Entity Framework”.

Siga os passos descritos nesse tópico e acrescente um modelo conceitual à aplicação. A partir desse ponto estamos prontos para acessar e manipular os dados do banco de dados Northwind.

Para invocar os métodos da interface `IGenericRepository` crie no método construtor do controlador uma nova instância da classe repositório `EntityFrameworkRepository` e passe ao método uma instância da classe ADO.NET Entity Framework `NorthwindBD` herdada de `ObjectContext`:

```
private IGenericRepository _repository;
public HomeController() {
    _repository = new EntityFrameworkRepository(new NorthwindBD());
}
```

Exemplo:

```
using System.Web.Mvc;
using AppCapitulo12.Models;

namespace AppCapitulo12.Controllers {
    public class HomeController : Controller {
        private IGenericRepository _repository;
        public HomeController() {
            _repository = new EntityFrameworkRepository(new NorthwindBD());
        }

        public ActionResult Index() {
            ViewBag.Message = "Welcome to ASP.NET MVC!";

            return View();
        }
        public ActionResult About() {
            return View();
        }
    }
}
```

Importe a namespace `AppCapitulo12.Models`:

```
using AppCapitulo12.Models;
```

A documentação do .NET Framework recomenda que os espaços para nome sejam nomeados de acordo com o seguinte critério:

```
<Empresa>.(<Produto>|<Tecnologia>)[.<Característica>][.<Subnamespace>]
```

Utilizar o nome da empresa ou até seu próprio nome evita que empresas ou desenvolvedores diferentes tenham namespaces com mesmo nome e prefixo.

Alguns exemplos de namespaces:

```
using Microsoft.CSharp;
using Microsoft.SqlServer;
using AlfredoLotar.Livro.AspNetMvc;
```

No segundo nível, temos o produto ou a tecnologia e, a partir do terceiro nível, a característica e as subnamespaces são opcionais.

Obs.: compile a aplicação conforme a figura 4.1.

12.6 Menu

O menu contém as categorias da tabela *Categories* do banco de dados *Northwind*.

Relendo o tópico “12.1.1 – Quais recursos devo usar”, concluímos que o menu é um candidato a cache, pois seu conteúdo é estático e compartilhado por quase todas as páginas do website.

Para exibir as categorias necessitamos:

- de um método de controlador;
- de um partial view;
- e carregar o partial view na página de layout.

Crie o método de controlador *Menu* que carrega o partial view *ViewMenu.cshtml*:

```
[ChildActionOnly, OutputCache(Duration = 60, VaryByParam = "none")]
public ActionResult Menu() {
    try {
        var model = _repository.GetAll<Category>();
        return PartialView("ViewMenu", model);
    }
    finally {
        if (_repository != null) _repository.Dispose();
    }
}
```

O partial view *Views/Home/ViewMenu.cshtml* contém uma tabela, associa a classe *Category* e carrega cada categoria em uma linha. Acrescente-o à aplicação com o código a seguir:

```
@model IEnumerable<AppCapitulo12.Models.Category>
<table style="width: 100%; border: 0">
    <tr>
        <td style="vertical-align: middle; background-color: #027399;">
            <div class="secao" style="text-align: left;">
```

```

        <strong>Categorias</strong>
    </div>
</td>
</tr>
@foreach (var item in Model) {
    <tr>
        <td>
            <div class="menu_lateral" style="text-align: left;">
                @Ajax.ActionLink(item.CategoryName, "ListaProdutos", new {
                    id = item.CategoryName }, new AjaxOptions { UpdateTargetId = "divProdutos" }
                )
            </div>
        </td>
    </tr>
}
</table>

```

Repare que os produtos de cada categoria são carregados via Ajax. Acrescente ao arquivo de layout Views\Shared_Layout.cshtml a linha a seguir:

```

<script src="@Url.Content("~/Scripts/jquery.unobtrusive-ajax.js")" type="text/javascript"></script>

```

O seletor CSS `menu_lateral` formata o menu:

```

.menu_lateral a {
    border-right: #ddd 1px solid;
    padding-right: 4px;
    border-top: #fff 1px solid;
    display: block;
    padding-left: 4px;
    font-size: 10px;
    background: #edf2f2;
    padding-bottom: 2px;
    color: #003399;
    text-indent: 1px;
    padding-top: 2px;
    border-bottom: #ddd 1px solid;
    font-family: arial, helvetica, sans-serif;
    text-decoration: none;
}

.menu_lateral a:hover {
    color: #000000;
    background-color: #e0e9e9;
}

```

Enquanto o seletor `secao` formata o título do menu:

```
.secao {
    padding-right: 4px;
    padding-left: 4px;
    font-size: 10px;
    padding-bottom: 2px;
    color: #ffffff;
    text-indent: 1px;
    padding-top: 2px;
    font-family: Arial, Helvetica, sans-serif;
    text-align: left;
}
```

Exclua do arquivo `Content/Site.css` os seletores CSS:

```
a:link {
    color: #034af3;
    text-decoration: underline;
}
a:visited {
    color: #505abc;
}
a:hover {
    color: #1d60ff;
    text-decoration: none;
}
a:active {
    color: #12eb87;
}
```

Obs.: insira o seletor CSS `menu_lateral` e `secao` no arquivo `Content/Site.css`.

Insira a linha a seguir na página de layout `Views/Shared/_Layout.cshtml`:

```
@{Html.RenderAction("Menu", "Home");}
```

Exemplo:

```
<table style="width: 100%; text-align: center; border: 0;">
    <tr>
        <td colspan="3" style="height: 10px;">
        </td>
    </tr>
    <tr>
        <td style="vertical-align: top; width: 15%">
            @{Html.RenderAction("Menu", "Home");}
        </td>
        <td style="vertical-align: top; width: 1%">
```



```

</td>
<td style="vertical-align: top; height: 800px; width: 80%">
    <table style="border-style: none; border-width: 0px; border-color: inherit;
        width: 100%;position: relative;">
        <tr>
            <td style="background-color: #027399;" colspan="3" class="secao">
                <strong>@ViewBag.Categoria</strong>
            </td>
        </tr>
        <tr>
            <td style="vertical-align: top; text-align: left;">
                @RenderBody()
            </td>
        </tr>
    </table>
</td>
</tr>
</table>

```

Obs.: se você tem dúvida onde inserir o menu, observe a figura 12.17.

Algumas URLs do menu requerem segmentos extras daqueles definidos no modelo.

Exemplo:

<http://localhost/Home/ListaProdutos/Grains/Cereals>

Na URL Cereals é o segmento extra.

Para permitir segmentos extras na URL do menu altere o trecho de código a seguir:

```

routes.MapRoute(
    "Default", // Route name
    "{controller}/{action}/{id}", // URL with parameters
    new { controller = "Home", action = "Index", id = UrlParameter.Optional } // Parameter defaults
);

```

Acrescente o caractere de asterisco antes do id:

```

routes.MapRoute(
    "Default", // Route name
    "{controller}/{action}/{*id}", // URL with parameters
    new { controller = "Home", action = "Index", id = UrlParameter.Optional } // Parameter defaults
);

```

12.6.1 Validação e formatação de dados

Quando você acessa as informações de determinada tabela, aproveite para validar e formatar os seus dados.

No exemplo, acessamos a tabela *Categories* do banco de dados *Northwind*.

Para validar as propriedades de uma classe do ADO.NET Entity Framework crie uma nova classe *partial* com o mesmo nome e sob a mesma namespace.

Para validar e formatar as propriedades da classe *Category*, crie o arquivo *Category.cs* no diretório *Models* e acrescente o código a seguir:

```
using System.ComponentModel.DataAnnotations;
namespace AppCapitulo12.Models {
    [MetadataType(typeof(CategoryMetadata))]
    public partial class Category { }

    public class CategoryMetadata {
        [Required(ErrorMessage = "O nome da categoria é obrigatório.")]
        [StringLength(15, MinimumLength = 3,
            ErrorMessage = "Digite no mínimo 3 e no máximo 15 caracteres.")]
        [RegularExpression(@"^[a-zA-Z0-9_\.\s]$",
            ErrorMessage = "O nome da categoria contém caracteres inválidos. Use a-zA-Z0-9_/ e espaços")]
        [Display(Name = "Categoria")]
        public string CategoryName { get; set; }

        [StringLength(150, MinimumLength = 0,
            ErrorMessage = "Digite no máximo 150 caracteres.")]
        [RegularExpression(@"^[a-zA-Z0-9_\.\s]$",
            ErrorMessage = "A descrição contém caracteres inválidos. Use a-zA-Z0-9_,. e espaços")]
        [Display(Name = "Descrição")]
        public string Description { get; set; }
    }
}
```

Obs.: garanta que a entrada contenha somente o conteúdo desejado. Utilize o acento circunflexo (^) e um cifrão (\$) como delimitadores no início e no fim de expressões regulares.

Para que a validação do lado do cliente funcione declare os arquivos a seguir na página de layout *Views/Shared/_Layout.cshtml*:

```
<script src="@Url.Content("~/Scripts/jquery.validate.min.js")" type="text/javascript"></script>
<script src="@Url.Content("~/Scripts/jquery.validate.unobtrusive.min.js")" type="text/javascript">
</script>
```

12.7 Exibindo os produtos

O conteúdo Ajax do website é exibido no view *Views/Home/Index.cshtml*. Substitua o código-padrão gerado pelo assistente:

```
@{
    ViewBag.Title = "Home Page";
}

<h2>@ViewBag.Message</h2>

<p>
    To learn more about ASP.NET MVC visit <a href="http://asp.net/mvc" title="ASP.NET MVC
    Website">http://asp.net/mvc</a>.
</p>
```

Por:

```
@{
    ViewBag.Title = "Programando com ASP.NET MVC";
}

<div id="divProdutos">
    @{Html.RenderAction("ListaProdutos", "Home");}
</div>
```

O método `ListaProdutos` exibe um produto de cada categoria na página inicial e os produtos de cada categoria quando clicamos em um item do menu. Adicione o método de controlador `ListaProdutos` ao controlador `HomeController`:

```
public ActionResult ListaProdutos(string id) {
    try {
        if (string.IsNullOrEmpty(id)) {
            ViewBag.Categoria = "Novidades";
            string sql = "SELECT * " + "FROM (SELECT p.ProductID, p.ProductName, p.SupplierID,
                p.CategoryID, p.QuantityPerUnit, " + "p.UnitPrice, p.UnitsInStock,
                p.UnitsOnOrder, p.Discontinued, p.ReorderLevel, c.CategoryName, " +
                "row_number() OVER (partition BY c.CategoryID ORDER BY ProductID DESC)
                AS ordem FROM dbo.Products AS p INNER JOIN dbo.Categories AS c
                ON p.CategoryID = c.CategoryID) AS TabelaDerivada " +
                "WHERE (TabelaDerivada.Ordem = 1)";

            var lst = new List<Product>();
            foreach (var item in _repository.ExecuteSQL<Product>(sql, null)) {
                lst.Add(new Product {ProductID=item.ProductID, ProductName = item.ProductName,
                    QuantityPerUnit = item.QuantityPerUnit, UnitPrice = item.UnitPrice,
                    UnitsInStock = item.UnitsInStock });
            }
            return PartialView("ViewProdutos", lst);
        }
        ViewBag.Categoria = id;
        var model = _repository.Find<Product>(p => p.Category.CategoryName == id);
        if (model == null)
            return View("NotFound");
        else
            return PartialView("ViewProdutos", model);
    }
}
```

```

    }
    finally {
        if (_repository != null) _repository.Dispose();
    }
}

```

Importe a namespace System.Collections.Generic:

```
using System.Collections.Generic;
```

Em ambos os casos, é carregado o partial view Views/Home/ViewProdutos.cshtml:

```

@model IEnumerable<AppCapitulo12.Models.Product>
<table style="border-style: none; border-width: 0px; border-color: inherit; width: 100%;
position: relative;">
    <tr>
        <td style="background-color: #027399; height: 0px;" colspan="3" class="secao">
            <strong>@ViewBag.Categoria</strong>
        </td>
    </tr>
    <tr>
        <td style="vertical-align: top; text-align: left;">
            @{
                if (Model.Count() > 0) {
                    WebGrid grid = new WebGrid(source: Model, canSort: false,
                        ajaxUpdateContainerId: "grid");
                    @grid.GetHtml(
                        htmlAttributes: new { id = "grid" },
                        tableStyle: "grade",
                        headerStyle: "header",
                        alternatingRowStyle: "alt",
                        columns: grid.Columns(
                            grid.Column("ProductName", header: "Produto"),
                            grid.Column("QuantityPerUnit", header: "Quant. por unidade"),
                            grid.Column("UnitPrice", header: "Preço",
                                format: @<text>@string.Format(
                                    new System.Globalization.CultureInfo("pt-BR"),
                                    "{0:c}", @item.UnitPrice)</text>),
                            grid.Column("UnitsInStock", header: "Unid. estoque"),
                            grid.Column(format: (item) => Html.ActionLink("Comprar",
                                "AddShoppingCartItem", "Shopping", new {
                                    ProductID = item.ProductID }, new { @class = "navegacao" })))
                        )
                }
            }
            else {
                @RenderPage("/Views/Shared/BuscaNaoEncontrado.cshtml");
            }
        </td>
    </tr>

```

```

    }
  </td>
</tr>
</table>

```

Crie o view Views/Shared/BuscaNaoEncontrado.cshtml:

```

@{
    Layout = null;
}
<!DOCTYPE html>
<html>
  <head>
    <title>Não encontrado</title>
  </head>
  <body>
    <div style="text-align: center;">
      <br />
      <h2>
        A busca não retornou nenhum registro.</p>
        Verifique a ortografia e tente novamente.
      </h2>
    </div>
  </body>
</html>

```

O WebGrid helper é formatado pelos seletores CSS a seguir:

```

.grade {
  font-family: Arial, Helvetica, sans-serif;
  font-size: 10px;
  margin: 0 px;
  border-collapse: collapse;
  width: 100%;
}
.header {
  font-family: Arial, Helvetica, sans-serif;
  font-size: 10px;
  background-color: #E8E8E8;
  font-weight: bold;
  color: Black;
  text-align: left;
}
.grade th, .grade td {
  border: 1px solid silver;
  padding: 5px;
  text-align: left;
}

```

```
.alt {
    background-color: #E8E8E8;
    color: #000;
}
```

Obs.: insira todos os seletores CSS usados neste capítulo no arquivo Content/Site.css.

12.7.1 Validação e formatação de dados

Nesta etapa da aplicação validamos e formatamos a tabela Products. Ao desenvolver código específico para determinada tabela, aproveite para validá-la, crie o arquivo Product.cs no diretório Models e acrescente o código a seguir:

```
using System.ComponentModel.DataAnnotations;
namespace AppCapítulo12.Models {
    [MetadataType(typeof(ProductMetadata))]
    public partial class Product { }

    public class ProductMetadata {
        [Required(ErrorMessage = "O nome do produto é obrigatório.")]
        [StringLength(40, MinimumLength = 3,
            ErrorMessage = "Digite no mínimo 3 e no máximo 40 caracteres.")]
        [RegularExpression(@"^[a-zA-ZÁ-Ú0-9_üÜß'\s-]+$",
            ErrorMessage = "O nome do produto contém caracteres inválidos.")]
        [Display(Name = "Produto")]
        public string ProductName { get; set; }

        [StringLength(20, MinimumLength = 0,
            ErrorMessage = "Digite no máximo 20 caracteres.")]
        [RegularExpression(@"^[a-zA-Z0-9_\s\.-]+$",
            ErrorMessage = "O campo Quant. por unidade contém caracteres inválidos.")]
        [Display(Name = "Quant. por unidade")]
        public string QuantityPerUnit { get; set; }

        [RegularExpression(@"^[0-9]{0,}$",
            ErrorMessage = "Digite um valor numérico positivo. Entre 0 e 922337203685477.")]
        [Range(typeof(decimal), "0", "922337203685477",
            ErrorMessage = "O campo Preço deve conter um valor numérico positivo. Entre 0 e 922337203685477.")]
        [DataType(DataType.Currency)]
        [Display(Name = "Preço")]
        public decimal UnitPrice { get; set; }

        [RegularExpression(@"^[0-9]{0,32767}$",
            ErrorMessage = "Digite um valor numérico positivo. Entre 0 e 32767.")]
        [Range(0, 32767, ErrorMessage = "O campo Unid. estoque deve conter um valor numérico positivo. Entre 0 e 32767.")]
    }
}
```

```

[Display(Name = "Unid. estoque")]
public short UnitsInStock { get; set; }

[RegularExpression("^([0-9]{0,32767})$",
    ErrorMessage = "Digite um valor numérico positivo. Entre 0 e 32767.")]
[Range(0, 32767, ErrorMessage = "O campo Unid. encomendadas deve conter um valor
numérico positivo. Entre 0 e 32767.")]
[Display(Name = "Unid. encomendadas")]
public short UnitsOnOrder { get; set; }

[RegularExpression("^([0-9]{0,32767})$",
    ErrorMessage = "Digite um valor numérico positivo. Entre 0 e 32767.")]
[Range(0, 32767, ErrorMessage = "O campo ReorderLevel deve conter um valor numérico
positivo. Entre 0 e 32767.")]
public short ReorderLevel { get; set; }

[Required(ErrorMessage = "O produto foi descontinuado. Sim/não?")]
[RegularExpression("^(true|false)+$",
    ErrorMessage = "O campo Descontinuado deve conter um valor booleano.")]
[Display(Name = "Descontinuado")]
public bool Discontinued { get; set; }
}
}

```

A validação foi adaptada para os dados contidos na tabela `Products`. Por exemplo, o campo `ProductName` contém informações com caracteres ü e ß.

Exemplo:

Original Frankfurter grüne Soße

A expressão regular valida ainda palavras com e sem acento, espaços em branco, números e o caractere de apóstrofo:

```
^[a-zä-üA-ZÁ-ÜÖ-9_üß'\s-]+$
```

12.8 Mecanismo de busca

O formulário de busca é exibido do lado esquerdo do website acima do menu. Para exibir o formulário de busca crie o partial view `Views/Home/Busca.cshtml` com o seguinte código:

```

<table style="border-color: #111111; width: 100%; background-color: #ffffff; border: 0;">
  <tr>
    <td style="width: 100%">
      <table style="border-color: #111111; width: 100%; border: 0;">
        <tr>
          <td style="background-color: #027399">

```

```

        <div class="secao" style="text-align: left">
            <strong>Pesquisar</strong>
        </div>
    </td>
</tr>
</table>
</td>
</tr>
<tr>
    <td style="height: 20px; background-color: #edf2f2;">
        @using (Ajax.BeginForm("Find", "Home", new AjaxOptions
        { UpdateTargetId = "divProdutos" })) {
            <table style="border-color: #111111; width: 100%; text-align: left; border: 0;">
                <tr>
                    <td style="text-align: left; width: 100%; background-color: #edf2f2;">
                        <table style="width: 100%; text-align: left;">
                            <tr style="text-align: center;">
                                <td>
                                    @Html.TextBox("txtBusca", "", new {
                                        MaxLength = 40, @class = "busca_input",
                                        @style = "text-align: left; width: 95%;" })
                                </td>
                                <td style="text-align: left; width: 5%;">
                                    <input type="submit" value="Ok" title="Procurar" alt="Ok" />
                                </td>
                            </tr>
                        </table>
                    </td>
                </tr>
            </table>
        }
    </td>
</tr>
<tr>
    <td style="width: 100%; background-color: #e4e4e4">
    </td>
</tr>
<tr>
    <td>
    </td>
</tr>
</table>

```

Repare que o seletor CSS `busca_input` formata a caixa de texto do formulário de busca:

```

.busca_input {
    border-right: #cfcfcf 1px solid;

```



```
border-top: #cfcfcf 1px solid;
font-size: 10px;
border-left: #cfcfcf 1px solid;
border-bottom: #cfcfcf 1px solid;
font-family: arial, helvetica, sans-serif;
height: 15px;
}
```

Em seguida, invoque o método de controlador `FormBusca` na página de layout `Views/Shared/_Layout.cshtml`:

```
<td style="vertical-align: top; width: 15%">
    @{Html.RenderAction("FormBusca", "Home");}
    @{Html.RenderAction("Menu", "Home");}
</td>
```

Exemplo:

```
<table style="width: 100%; text-align: center; border: 0;">
    <tr>
        <td colspan="3" style="height: 20px;">
        </td>
    </tr>
    <tr>
        <td style="vertical-align: top; width: 15%">
            @{Html.RenderAction("FormBusca", "Home");}
            @{Html.RenderAction("Menu", "Home");}
        </td>
        <td style="vertical-align: top; width: 1%">
        </td>
        <td style="vertical-align: top; height: 800px; width: 80%">
            <table style="border-style: none; border-width: 0px; border-color: inherit;
                width: 100%; position: relative;">
                <tr>
                    <td style="background-color: #027399;" colspan="3" class="secao">
                        <strong>@ViewBag.Categoria</strong>
                    </td>
                </tr>
                <tr>
                    <td style="vertical-align: top; text-align: left;">
                        @RenderBody()
                    </td>
                </tr>
            </table>
        </td>
    </tr>
</table>
```

No controlador `HomeController` acrescentamos o método de controlador `FormBusca` responsável por carregar o formulário `Views/Home/Busca.cshtml`:

```
[ChildActionOnly]
public ActionResult FormBusca() {
    return PartialView("Busca");
}
```

E o método `Find` responsável por processar a busca e carregar o partial view `Views/Home/ViewProdutos.cshtml` com o resultado da busca:

```
public ActionResult Find(string txtBusca) {
    try {
        string termo = string.Empty;
        if (!string.IsNullOrEmpty(txtBusca)) {
            Response.Cookies["Busca"].Value = txtBusca.Trim();
            termo = txtBusca.Trim();
        }
        else {
            if (Request.Cookies["Busca"] != null) termo = Request.Cookies["Busca"].Value;
        }
        if (!Regex.IsMatch(termo.Trim(), @"^[a-zá-úA-ZÁ-Ú0-9_üß'\s-]{1,40}$")) return View();

        var model = _repository.Find<Product>(p => p.ProductName.Contains(termo.Trim()));
        if (model == null)
            return PartialView("NotFound");

        ViewBag.Categoria = "Resultado da busca para: " + Server.HtmlEncode(termo.Trim());
        return PartialView("ViewProdutos", model);
    }
    finally {
        if (_repository != null) _repository.Dispose();
    }
}
```

A validação é realizada por meio de expressões regulares:

```
if (!Regex.IsMatch(termo.Trim(), @"^[a-zá-úA-ZÁ-Ú0-9_üß'\s-]{1,40}$")) return View();
```

Para que o mecanismo de paginação funcione usamos cookies:

```
string termo = string.Empty;
if (!string.IsNullOrEmpty(txtBusca)) {
    Response.Cookies["Busca"].Value = txtBusca.Trim();
    termo = txtBusca.Trim();
}
```

Importe a namespace `System.Text.RegularExpressions`:

```
using System.Text.RegularExpressions;
```

12.9 Formulário de contato

O formulário de contato é extremamente simples. Possui quatro campos (Nome, Email, Assunto, Mensagem). Envia por e-mail e grava as informações em uma tabela do banco de dados Northwind.

O formulário de contato é carregado por meio de um link na página de layout.

O formulário de contato invoca o método `AddContato` e adiciona as informações do contato no banco de dados. Observe a figura 12.18.

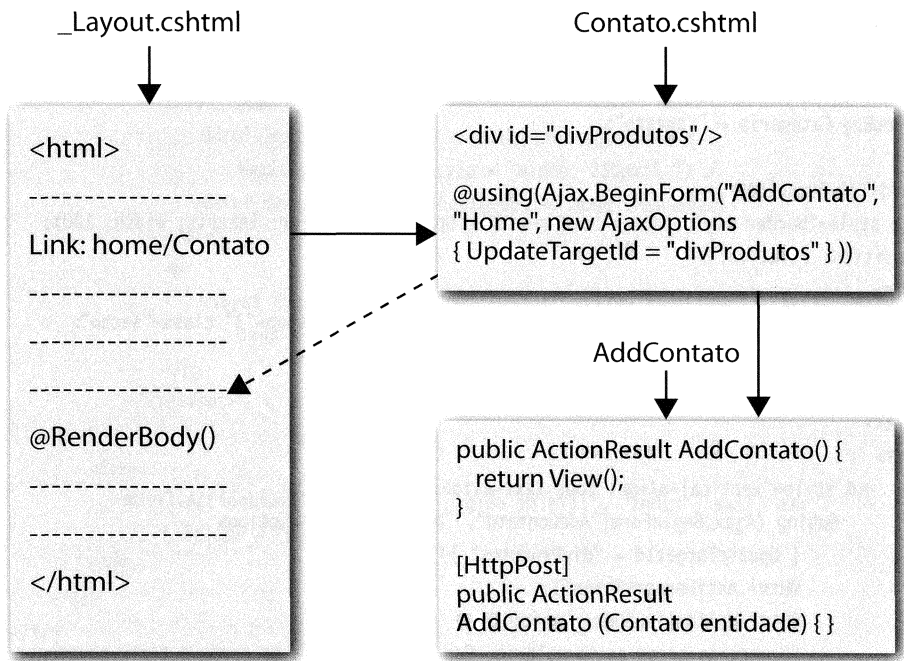


Figura 12.18 – Relação entre os arquivos e o método `addcontato`.

12.9.1 O banco de dados

Acrescente a tabela `Contatos` ao banco de dados Northwind e os campos a seguir:

Campo	Tipo de dados	Permite nulos	Especificação de identidade
ContatoId	Int	Não	Sim
Nome	varchar(50)	Não	
Email	nvarchar(50)	Não	
Assunto	nvarchar(50)	Não	
Mensagem	nvarchar(300)	Não	

Em seguida, atualize o modelo conceitual do ADO.NET Entity Framework. Abra o arquivo `Models/Model1.edmx` e clique com o botão direito do mouse em uma área livre do arquivo. Em seguida, selecione **Update Model from Database....** Adicione a tabela Contatos.

12.9.2 Projetando o formulário de contato

O formulário de contato é concebido com o auxílio de Html helpers. As mensagens de erro são exibidas ao lado dos campos em letras vermelhas.

Adicione o arquivo `Views/Home/Contato.cshtml` à aplicação e acrescente o código a seguir:

```
@model AppCapitulo12.Models.Contato
@{
    ViewBag.Title = "Programando com ASP.NET MVC";
    ViewBag.Categoria = "Contato";
}
<div id="divProdutos" />
<table style="border-style: none; border-width: 0px; border-color: inherit; width: 100%;
    position: relative;">
    <tr>
        <td style="background-color: #027399; height: 0px;" colspan="3" class="secao">
            <strong>@ViewBag.Categoria</strong>
        </td>
    </tr>
    <tr>
        <td style="vertical-align: top; text-align: left;">
            @using (Ajax.BeginForm("AddContato", "Home", new AjaxOptions
                { UpdateTargetId = "divProdutos" })) {
                @Html.AntiForgeryToken()
                @Html.ValidationSummary(true,
                    "Ocorreram erros no formulário. Por favor, corrija os erros e tente novamente.");
                <fieldset>
                    <div class="editor-label">
                        @Html.LabelFor(model => model.Nome)
                    </div>
                    <div class="editor-field">
                        @Html.TextBoxFor(model => model.Nome, new {
                            MaxLength = 50, @style = "width: 250px;" })
                        @Html.ValidationMessageFor(model => model.Nome)
                    </div>
                    <div class="editor-label">
                        @Html.LabelFor(model => model.Email)
                    </div>
                    <div class="editor-field">
                        @Html.TextBoxFor(model => model.Email, new {
                            MaxLength = 50, @style = "width: 250px;" })
                    </div>
                </fieldset>
            }
        </td>
    </tr>
</table>
```

```

        @Html.ValidationMessageFor(model => model.Email)
    </div>
    <div class="editor-label">
        @Html.LabelFor(model => model.Assunto)
    </div>
    <div class="editor-field">
        @Html.TextBoxFor(model => model.Assunto, new {
            MaxLength = 50, @style = "width: 250px;" })
        @Html.ValidationMessageFor(model => model.Assunto)
    </div>
    <div class="editor-label">
        @Html.LabelFor(model => model.Mensagem)
    </div>
    <div class="editor-field">
        @Html.TextAreaFor(model => model.Mensagem, new {
            MaxLength = 300, @style = "width: 250px;" })
        @Html.ValidationMessageFor(model => model.Mensagem)
    </div>
    <p>
        <input type="submit" value="Enviar" style="font-family: Arial, Helvetica,
            sans-serif;font-size: 10px;" />
    </p>
</fieldset>
}
<div>
    @Html.ActionLink("Volta para a página inicial", "Index", null, new
        { @class = "navegacao" })
</div>
</td>
</tr>
</table>

```

Acrescente aos seletores CSS text-box, editor-field, editor-label, input[type="text"], input[type="password"] e textarea localizados no arquivo Content/Site.css a linha a seguir:

font-family: Arial, Helvetica, sans-serif; font-size: 10px;

A saída gerada vemos na figura 12.19.

O arquivo Views/Home/Contato.cshtml é invocado por um link não Ajax. Para que os links Ajax da página não sejam desativados inserimos a linha anterior no partial view contato.

```
<div id="divProdutos"/>
```

Lembre-se de que essa abordagem somente é necessária quando você usa links Ajax e um formulário invocado por um link não Ajax. A tag div com id igual a divProdutos é usada pela página de layout para carregar os produtos, exibir o formulário de contato, o carrinho de compras etc.

The image shows a web form titled "Contato". It has four text input fields labeled "Nome", "E-mail", "Assunto", and "Mensagem". Below these fields is a button labeled "Enviar". At the bottom of the form, there is a link that says "Volta para a página inicial".

Figura 12.19 – Formulário de contato.

O formulário de contato invoca o método `AddContato` via Ajax:

```
Ajax.BeginForm("AddContato", "Home", new AjaxOptions { UpdateTargetId = "divProdutos" }))) {
```

A linha:

```
@Html.AntiForgeryToken()
```

Juntamente com o atributo `ValidateAntiForgeryToken` protege o formulário contra ataques de um único clique (CSRF).

12.9.3 O método `AddContato`

O método `AddContato` grava no banco de dados as informações digitadas no formulário pelo usuário. No controlador `HomeController` temos dois formulários `AddContato`. O primeiro exibe o formulário em branco na tela e define o texto da seção:

```
public ActionResult AddContato() {
    Response.Cache.SetCacheability(HttpCacheability.NoCache);
    Response.Cache.SetAllowResponseInBrowserHistory(false);
    ViewBag.Categoria = "Contato";
    return View();
}
```

As linhas:

```
Response.Cache.SetCacheability(HttpCacheability.NoCache);
Response.Cache.SetAllowResponseInBrowserHistory(false);
```

Evitam que o formulário seja postado várias vezes com o mesmo conteúdo:

O segundo método `AddContato` envia os dados para o banco de dados e via e-mail:

[HttpPost, ValidateAntiForgeryToken]

```
public ActionResult AddContato([Bind(Exclude = "ContatoId")])Contato entidade) {
    try {
        if (!ModelState.IsValid) {
            ModelState.AddModelError("Error_Form",
                "Ocorreram erros no formulário. Por favor, corrija os erros e tente novamente. ");
            return View();
        }
        else {
            _repository.Create<Contato>(entidade);

            SendEmail(entidade.Email, "Contato pelo website.",
                "Obrigado por entrar em contato conosco.<p>Entraremos em contato em breve.</p>");
            return PartialView("EnviadoComSucesso");
        }
    }
    catch (System.Data.EntitySqlException) {
        ModelState.AddModelError("Error_Form", "Erro EntitySqlException. ");
        return View();
    }
    catch (System.Data.EntityCommandExecutionException) {
        ModelState.AddModelError("Error_Form", "Erro EntityCommandExecutionException.");
        return View();
    }
    finally
    {
        if (_repository != null) _repository.Dispose();
    }
}
```

Crie o view `Views/Shared/EnviadoComSucesso.cshtml` com o conteúdo a seguir:

```
<h2>Informações enviadas com sucesso.</h2>
<p />
@Html.ActionLink("Volta para a página inicial.", "index", "Home", null,
    new { @class = "navegacao" })
```

O método `SendEmail` envia um e-mail ao usuário após gravar as informações no banco de dados:

```
private void SendEmail(string email, string assunto, string mensagem) {
    try {
        WebMail.SmtpServer = "localhost";
        WebMail.SmtpPort = 25;
```

```

        //WebMail.EnableSsl = true;
        //WebMail.UserName = "usuário";
        //WebMail.Password = "senha";
        WebMail.From = "siteexemplo@siteexemplo.com";
        WebMail.SmtpUseDefaultCredentials = true;

        WebMail.Send(to: email, subject: assunto, body: mensagem);
    }
    catch (Exception ex) {
        ModelState.AddModelError("Error_Form", ex.Message);
        //ModelState.AddModelError("", "Erro ao enviar e-mail.");
    }
}

```

Importe as namespaces a seguir:

```

using System.Web;
using System.Web.Helpers;

```

Acrescente também o método `HandleUnknownAction`:

```

protected override void HandleUnknownAction(string actionName) {
    try {
        this.View(actionName).ExecuteResult(this.ControllerContext);
    }
    catch (InvalidOperationException) {
        this.View("NotFound").ExecuteResult(this.ControllerContext);
    }
}

```

Crie o view `Views/Home/NotFound.cshtml`:

```
<p>Recurso ou item não encontrado.</p>
```

12.9.4 Validação e formatação

Para formatar e validar as informações da tabela `Contatos` criamos o arquivo `Contato.cs` no diretório `Models`.

Para validar as propriedades de uma classe do ADO.NET Entity Framework crie uma nova classe parcial com o mesmo nome e sob a mesma namespace:

```

using System.ComponentModel.DataAnnotations;
namespace AppCapitulo12.Models {
    [MetadataType(typeof(ContatoMetadata))]
    public partial class Contato { }
}

```



```

public class ContatoMetadata {
    [Required(ErrorMessage = "O nome é obrigatório. ")]
    [StringLength(50, MinimumLength = 1,
        ErrorMessage = "Digite no mínimo 1 e no máximo 50 caracteres. ")]
    [RegularExpression(@"^[a-zA-ZÁ-Ú0-9_ü'\s]+$",
        ErrorMessage = "O nome contém caracteres inválidos. Use a-zá-úa-ZÁ-Ú0-9_ü e espaços")]
    public string Nome { get; set; }

    [Required(ErrorMessage = "O e-mail é obrigatório.")]
    [StringLength(50, MinimumLength = 1,
        ErrorMessage = "Digite no mínimo 1 e no máximo 50 caracteres.")]
    [RegularExpression(@"^\w+([-+.] \w+)*@\w+([-.] \w+)*\.\w+([-.] \w+)*$",
        ErrorMessage = "Digite um e-mail válido.")]
    [Display(Name = "E-mail")]
    public string Email { get; set; }

    [Required(ErrorMessage = "O assunto é obrigatório.")]
    [StringLength(50, MinimumLength = 1,
        ErrorMessage = "Digite no mínimo 1 e no máximo 50 caracteres.")]
    [RegularExpression(@"^[a-zA-ZÁ-Ú0-9_ü'\s]+$",
        ErrorMessage = "O assunto contém caracteres inválidos. Use a-zá-úa-ZÁ-Ú0-9_ü e espaços")]
    public string Assunto { get; set; }

    [Required(ErrorMessage = "A mensagem é obrigatória.")]
    [StringLength(300, MinimumLength = 1,
        ErrorMessage = "Digite no mínimo 1 e no máximo 300 caracteres.")]
    [RegularExpression(@"^[a-zA-ZÁ-Ú0-9_ü'\.!\?\\, \s-]+$", ErrorMessage = "O nome da
categoria contém caracteres inválidos. Use a-zá-úa-ZÁ-Ú0-9_ü'.!?, e espaços")]
    public string Mensagem { get; set; }
}
}

```

12.10 Cadastro de usuários

O cadastro de usuários é realizado em duas etapas. A primeira solicita informações de login e a segunda etapa, as informações pessoais, como o nome, o endereço, o CPF etc.

Cada etapa do cadastro necessita de um formulário e também de uma tabela, ou seja, temos dois formulários e duas tabelas.

Observe as figuras 12.20 e 12.21.

Figura 12.20 – Formulários de cadastro de login.

Figura 12.21 – Formulários de cadastro de dados pessoais.

Quando criamos um nova aplicação ASP.NET MVC, o Visual Studio cria um formulário de cadastro `Views/Account/Register.cshtml` com as informações login.

Traduza os textos em inglês do formulário `Views/Account/Register.cshtml`. Exemplo:

```
@Html.ValidationSummary(true, "Não foi possível criar a nova conta. Por favor, verifique os erros a seguir.")
```

Remova as linhas:

```
<h2>Create a New Account</h2>
```

```
<p>
```

```
    Use the form below to create a new account.
```

```
</p>
```

```
<p>
```

```
    Passwords are required to be a minimum of @ViewBag.PasswordLength characters in length.
```

```
</p>
```

```
<legend>Informações da conta</legend>
```

```
<script src="@Url.Content("~/Scripts/jquery.validate.min.js")" type="text/javascript"></script>
```

```
<script src="@Url.Content("~/Scripts/jquery.validate.unobtrusive.min.js")"
    type="text/javascript"></script>
```

Defina o título da seção:

```
@{
    ViewBag.Categoria = "Crie uma nova conta";
}
```

Evite que links Ajax sejam desativados. Acrescente a linha:

```
<div id="divProdutos" />
```

Insira o HTML helper AntiForgeryToken:

```
@Html.AntiForgeryToken()
```

Altere a fonte e o texto do botão de comando:

```
<p>
    <input type="submit" value="Registrar" style="font-family: Arial, Helvetica, sans-serif;
        font-size: 10px;" />
</p>
```

Observe o resultado a seguir:

```
@model AppCapitulo12.Models.RegisterModel
@{
    ViewBag.Title = "Programando com ASP.NET MVC";
    ViewBag.Categoria = "Crie uma nova conta";
}
<div id="divProdutos" />
<table style="border-style: none; border-width: 0px; border-color: inherit;
    width: 100%;position: relative;">
    <tr>
        <td style="background-color: #027399; height: 0px;" colspan="3" class="secao">
            <strong>@ViewBag.Categoria</strong>
        </td>
        <tr>
            <td style="vertical-align: top; text-align: left;">
                @using (Html.BeginForm()) {
                    @Html.ValidationSummary(true,
                        "Não foi possível criar a nova conta. Por favor, verifique os erros a seguir.")
                    @Html.AntiForgeryToken()
                    <div>
                        <fieldset>
                            <div class="editor-label">
                                @Html.LabelFor(m => m.UserName)
                            </div>
```

```

        <div class="editor-field">
            @Html.TextBoxFor(m => m.UserName, new { MaxLength = 50 })
            @Html.ValidationMessageFor(m => m.UserName)
        </div>
        <div class="editor-label">
            @Html.LabelFor(m => m.Email)
        </div>
        <div class="editor-field">
            @Html.TextBoxFor(m => m.Email)
            @Html.ValidationMessageFor(m => m.Email)
        </div>
        <div class="editor-label">
            @Html.LabelFor(m => m.Password)
        </div>
        <div class="editor-field">
            @Html.PasswordFor(m => m.Password, new { MaxLength = 15 })
            @Html.ValidationMessageFor(m => m.Password)
        </div>
        <div class="editor-label">
            @Html.LabelFor(m => m.ConfirmPassword)
        </div>
        <div class="editor-field">
            @Html.PasswordFor(m => m.ConfirmPassword, new { MaxLength = 15 })
            @Html.ValidationMessageFor(m => m.ConfirmPassword)
        </div>
        <p>
            <input type="submit" value="Registrar"
                style="font-family: Arial, Helvetica, sans-serif;font-size: 10px;" />
        </p>
    </fieldset>
</div>
    }
</td>
</tr>
</table>

```

O formulário já foi formatado; então, prossiga e acrescente as linhas a seguir ao método Register da classe controladora AccountController:

```

Roles.AddUserToRole(model.UserName, "Clientes");
return RedirectToAction("Create", "Account");

```

Exemplo:

```

[HttpPost, ValidateAntiForgeryToken]//,RequireHttps
public ActionResult Register(RegisterModel model) {
    if (ModelState.IsValid) {

```

```
// Attempt to register the user
MembershipCreateStatus createStatus =
    MembershipService.CreateUser(model.UserName, model.Password, model.Email);

if (createStatus == MembershipCreateStatus.Success) {
    FormsService.SignIn(model.UserName, false /* createPersistentCookie */);
    Roles.AddUserToRole(model.UserName, "Clientes");
    return RedirectToAction("Create", "Account");
}
else {
    ModelState.AddModelError("", AccountValidation.ErrorCodeToString(createStatus));
}
}
```

```
// If we got this far, something failed, redisplay form
ViewBag.PasswordLength = MembershipService.MinPasswordLength;
return View(model);
}
```

O novo usuário é adicionado ao grupo Clientes:

```
Roles.AddUserToRole(model.UserName, "Clientes");
```

Antes de adicionar um usuário crie o grupo Clientes no banco de dados conforme instruções do tópico “11.4 Gerenciando a aplicação”.

Em seguida, o usuário já logado é redirecionado para o segundo formulário de cadastro.

```
return RedirectToAction("Create", "Account");
```

Obs.: não habilitamos o atributo RequireHttps para facilitar aos leitores a execução da aplicação.

12.10.1 Validando a entrada

Valide as propriedades da classe RegisterModel no arquivo Models/AccountModels.cs conforme código a seguir:

```
public class RegisterModel {
    [Required(ErrorMessage = "Nome do usuário é obrigatório.")]
    [RegularExpression("[a-zA-Z0-9]{1,256}$", ErrorMessage = "Digite um nome de usuário válido. Não use espaços. Somente letra e números.")]
    [Display(Name = "Usuário")]
    public string UserName { get; set; }

    [Required(ErrorMessage = "E-mail é obrigatório.")]
    [DataType(DataType.EmailAddress)]
}
```

```

[RegularExpression(@"^\w+([-+.]\\w+)*@\w+([-+.]\\w+)*\.\\w+([-+.]\\w+)*$",
    ErrorMessage = "Digite um e-mail válido.")]
[Display(Name = "E-mail")]
public string Email { get; set; }

[Required(ErrorMessage = "Senha é obrigatória.")]
[ValidatePasswordLength]
[RegularExpression(@"^[a-zA-Z0-9_\\*\\-\\+\\.\\?\\!]{8,15}$", ErrorMessage = "Digite uma senha
válida. Use letras, números e os caracteres: *-+.?!")
[DataType(DataType.Password)]
[Display(Name = "Senha")]
public string Password { get; set; }

[Required(ErrorMessage = "Senha de confirmação é obrigatória.")]
[DataType(DataType.Password)]
[RegularExpression(@"^[a-zA-Z0-9_\\*\\-\\+\\.\\?\\!]{8,15}$", ErrorMessage = "Digite uma senha
válida. Use letras, números e os caracteres: *-+.?!")
[Display(Name = "Confirme a senha")]
[Compare("Password", ErrorMessage = "A nova senha não coincide com a senha de confirmação.
Por favor, digite novamente.")]
public string ConfirmPassword { get; set; }
}

```

12.10.2 Criando a tabela

A tabela de cadastro de usuários deve conter um campo que associa as informações pessoais às informações de login. Por exemplo, o campo `UserId`.

Crie no banco de dados Northwind a tabela `Cadastros`. Acrescente os campos conforme lista a seguir:

Campo	Tipo de dados	Permite nulos	Especificação de identidade
<code>CadastroId</code>	<code>Int</code>	Não	Sim
<code>UserId</code>	<code>uniqueidentifier</code>	Não	
<code>Nome</code>	<code>varchar(50)</code>	Não	
<code>Cpf</code>	<code>char(15)</code>	Não	
<code>Endereco</code>	<code>nvarchar(100)</code>	Não	
<code>Cep</code>	<code>char(9)</code>	Não	
<code>Cidade</code>	<code>varchar(50)</code>	Não	
<code>Estado</code>	<code>char(2)</code>	Não	
<code>Completo</code>	<code>bit</code>	Não	

O campo `Completo` pode ser usado para monitorar as etapas de preenchimento do formulário. Quando o usuário preencheu e enviou as informações dos dois formulários, o campo `Completo` é igual `true`, ou seja, o usuário possui um cadastro completo.

12.10.3 ADO.NET Entity Framework

Atualize o modelo conceitual do ADO.NET Entity Framework. Abra o arquivo `Models/Model1.edmx` e clique com o botão direito do mouse em uma área livre do arquivo. Em seguida, selecione **Update Model from Database...** Adicione a tabela `Cadastros`.

12.10.4 Validando as propriedades

Valide as propriedades da tabela `Cadastros`. Crie a classe `Cadastro.cs` no diretório `Models`.

Para validar as propriedades de uma classe do ADO.NET Entity Framework crie uma nova classe parcial com o mesmo nome e sob a mesma namespace:

```
using System.ComponentModel.DataAnnotations;
namespace AppCapitulo12.Models {
    [MetadataType(typeof(CadastroMetadata))]
    public partial class Cadastro { }

    public class CadastroMetadata {
        [Required(ErrorMessage = "O nome é obrigatório.")]
        [StringLength(50, MinimumLength = 1,
            ErrorMessage = "Digite no mínimo 1 e no máximo 50 caracteres.")]
        [RegularExpression(@"^[a-zA-ZÁ-Ú0-9_üÜ'\s]+$",
            ErrorMessage = "O nome contém caracteres inválidos. Use a-zá-úa-ZÁ-ú0-9_üÜ e espaços")]
        public string Nome { get; set; }

        [Required(ErrorMessage = "O campo Cpf é obrigatório.")]
        [RegularExpression(@"^\d{2}.\d{2}.\d{2}-\d{2}?$",
            ErrorMessage = "Cpf inválido. Ex.: 00.00.00.000-00")]
        public string Cpf { get; set; }

        [Required(ErrorMessage = "O endereço é obrigatório.")]
        [StringLength(100, MinimumLength = 1,
            ErrorMessage = "Digite no máximo 100 caracteres.")]
        [RegularExpression(@"^[a-zA-ZÁ-Ú0-9_üÜ'\.\s]+$", ErrorMessage = "O campo Endereço contém caracteres inválidos. Use a-zá-úa-ZÁ-ú0-9_üÜ. e espaços")]
        [Display(Name = "Endereço")]
        public string Endereco { get; set; }

        [Required(ErrorMessage = "O campo Cep é obrigatório.")]
        [RegularExpression(@"^\d{5}-\d{3}?$", ErrorMessage = "Cep inválido.")]
        public string Cep { get; set; }

        [Required(ErrorMessage = "O campo Cidade é obrigatório.")]
        [StringLength(50, MinimumLength = 1, ErrorMessage = "Digite no máximo 50 caracteres.")]
        [RegularExpression(@"^[a-zA-ZÁ-Ú0-9_üÜ'\.\s]+$", ErrorMessage = "O campo Cidade contém caracteres inválidos. Use a-zá-úa-ZÁ-ú0-9_üÜ. e espaços")]
    }
}
```

```

    public string Cidade { get; set; }

    [RegularExpression("[a-zA-Z]{2}$",
        ErrorMessage = "O campo Estado contém caracteres inválidos")]
    public string Estado { get; set; }
}
}

```

12.10.5 Métodos de controlador

No controlador `AccountController` acrescente as linhas a seguir:

```

private IGenericRepository _repository;
public AccountController() {
    _repository = new EntityFrameworkRepository(new NorthwindBD());
}

```

Acrescente também o método `HandleUnknownAction`:

```

protected override void HandleUnknownAction(string actionName) {
    try {
        this.View(actionName).ExecuteResult(this.ControllerContext);
    }
    catch (InvalidOperationException) {
        this.View("NotFound").ExecuteResult(this.ControllerContext);
    }
}

```

Crie também dois métodos. Um que carrega o formulário em branco:

```

public ActionResult Create() {
    if (!User.Identity.IsAuthenticated) return RedirectToAction("Register", "Account");
    return View();
}

```

O segundo envia as informações para a tabela `Cadastros` do banco de dados `Northwind`:

```

[HttpPost, ValidateAntiForgeryToken]//, RequireHttps
public ActionResult Create([Bind(Exclude = "CadastroId, Completo, UserId")]Cadastro entidade) {
    try {
        if (!ModelState.IsValid) {
            ModelState.AddModelError("Error_Form",
                "Ocorreram erros no formulário. Por favor, corrija os erros e tente novamente.");
            return View();
        }
        else {
            if (User.Identity.IsAuthenticated) {
                MembershipUser usuario = Membership.GetUser(User.Identity.Name);
                entidade.UserId = new Guid(usuario.ProviderUserKey.ToString());
            }
        }
    }
}

```



```

        entidade.Completo = true;
        _repository.Create<Cadastro>(entidade);
        return PartialView("EnviadoComSucesso");
    }
    else {
        return RedirectToAction("Index", "Home");
    }
}
}
}
catch (System.Data.EntitySqlException) {
    ModelState.AddModelError("Error_Form", "Erro EntitySqlException.");
    return View();
}
catch (System.Data.EntityCommandExecutionException) {
    ModelState.AddModelError("Error_Form", "Erro EntityCommandExecutionException.");
    return View();
}
finally {
    if (_repository != null) _repository.Dispose();
}
}
}

```

Antes de enviar as informações pessoais do usuário para o banco de dados, verificamos se o usuário está logado:

```
if (User.Identity.IsAuthenticated) {
```

Capturamos o nome de usuário e o UserId:

```
MembershipUser usuario = Membership.GetUser(User.Identity.Name);
entidade.UserId = new Guid(usuario.ProviderUserKey.ToString());
```

Definimos o formulário como completamente preenchido:

```
entidade.Completo = true;
```

Enviamos as informações para o banco de dados e exibimos a mensagem de confirmação:

```
_repository.Create<Cadastro>(entidade);
return PartialView("EnviadoComSucesso");
```

12.10.6 Projetando o formulário de cadastro

O formulário de cadastro solicita ao usuário informações pessoais, como nome, CPF, endereço, CEP, cidade, estado.

Crie o arquivo `Create.cshtml` no diretório `Account`.

Acrescente o código a seguir:

```
@model AppCapitulo12.Models.Cadastro
@{
    ViewBag.Title = "Programando com ASP.NET MVC";
    ViewBag.Categoria = "Cadastro de usuários - continuação ...";
}
<div id="divProdutos" />
<table style="border-style: none; border-width: 0px; border-color: inherit;
width: 100%;position: relative;">
    <tr>
        <td style="background-color: #027399; height: 0px;" colspan="3" class="secao">
            <strong>@ViewBag.Categoria</strong>
        </td>
    </tr>
    <tr>
        <td style="vertical-align: top; text-align: left;">
            @using (Ajax.BeginForm("Create", "Account",
                new AjaxOptions { UpdateTargetId = "divProdutos" })) {
                @Html.ValidationSummary(true)
                @Html.AntiForgeryToken()
                <fieldset>
                    <div class="editor-label">
                        @Html.LabelFor(model => model.Nome)
                    </div>
                    <div class="editor-field">
                        @Html.TextBoxFor(model => model.Nome,
                            new { MaxLength = 50, @style = "width: 260px;" })
                        @Html.ValidationMessageFor(model => model.Nome)
                    </div>
                    <div class="editor-label">
                        @Html.LabelFor(model => model.Cpf)
                    </div>
                    <div class="editor-field">
                        @Html.TextBoxFor(model => model.Cpf,
                            new { MaxLength = 15, @style = "width: 90px;" })
                        @Html.ValidationMessageFor(model => model.Cpf)
                    </div>
                    <div class="editor-label">
                        @Html.LabelFor(model => model.Endereco)
                    </div>
                    <div class="editor-field">
                        @Html.TextBoxFor(model => model.Endereco,
                            new { MaxLength = 100, @style = "width: 230px;" })
                        @Html.ValidationMessageFor(model => model.Endereco)
                    </div>
                </fieldset>
            }
        </td>
    </tr>
</table>
```

```
<div class="editor-label">
    @Html.LabelFor(model => model.Cep)
</div>
<div class="editor-field">
    @Html.TextBoxFor(model => model.Cep,
        new { MaxLength = 9, @style = "width: 50px;" })
    @Html.ValidationMessageFor(model => model.Cep)
</div>
<div class="editor-label">
    @Html.LabelFor(model => model.Cidade)
</div>
<div class="editor-field">
    @Html.TextBoxFor(model => model.Cidade,
        new { MaxLength = 50, @style = "width: 200px;" })
    @Html.ValidationMessageFor(model => model.Cidade)
</div>
<div class="editor-label">
    @Html.LabelFor(model => model.Estado)
</div>
<div class="editor-field">
    @Html.DropDownListFor(model => model.Estado, new[] {
        new SelectListItem { Text = "AL", Value = "AL" },
        new SelectListItem { Text = "AM", Value = "AM" },
        new SelectListItem { Text = "AP", Value = "AP" },
        new SelectListItem { Text = "BA", Value = "BA" },
        new SelectListItem { Text = "CE", Value = "CE" },
        new SelectListItem { Text = "DF", Value = "DF" },
        new SelectListItem { Text = "ES", Value = "ES" },
        new SelectListItem { Text = "GO", Value = "GO" },
        new SelectListItem { Text = "MA", Value = "MA" },
        new SelectListItem { Text = "MG", Value = "MG" },
        new SelectListItem { Text = "MS", Value = "MS" },
        new SelectListItem { Text = "MT", Value = "MT" },
        new SelectListItem { Text = "PA", Value = "PA" },
        new SelectListItem { Text = "PB", Value = "PB" },
        new SelectListItem { Text = "PR", Value = "PR" },
        new SelectListItem { Text = "PE", Value = "PE" },
        new SelectListItem { Text = "PI", Value = "PI" },
        new SelectListItem { Text = "RJ", Value = "RJ" },
        new SelectListItem { Text = "RN", Value = "RN" },
        new SelectListItem { Text = "RS", Value = "RS", Selected=true },
        new SelectListItem { Text = "RR", Value = "RR" },
        new SelectListItem { Text = "RO", Value = "RO" },
        new SelectListItem { Text = "SC", Value = "SC" },
        new SelectListItem { Text = "SP", Value = "SP" },
```

```

        new SelectListItem { Text = "SE", Value = "SE" },
        new SelectListItem { Text = "TO", Value = "TO" }
    }, null, new { @class = "editor-field" })
    @Html.ValidationMessageFor(model => model.Estado)
</div>
<p>
    <input type="submit" value="Enviar cadastro"
    style="font-family: Arial, Helvetica, sans-serif;font-size: 10px;" />
</p>
</fieldset>
}
<div>
    @Html.ActionLink("Volta para a página inicial", "Index", "home", null,
    new { @class = "navegacao" })
</div>
</td>
</tr>
</table>

```

Observe o título da seção:

```

@{
    ViewBag.Categoria = "Cadastro de usuários - continuação ...";
}

```

E a já famosa tag div:

```
<div id="divProdutos" />
```

O atributo `MaxLength` dos campos do formulário controla a quantidade de caracteres permitida:

```
@Html.TextBoxFor(model => model.Nome, new { MaxLength = 50, @style = "width: 260px;" })
```

12.10.7 Informações de criptografia

Informações sigilosas devem ser criptografadas quando armazenadas no banco de dados. Use um algoritmo de criptografia simétrico, como `Rijndael`.

Para criptografar dados confidenciais gere uma chave de criptografia com a classe `RNGCryptoServiceProvider`:

```

RNGCryptoServiceProvider rng = new RNGCryptoServiceProvider();
byte[] key = new byte[16];
rng.GetBytes(key);
string chave = Convert.ToBase64String(key);

```

Faça backup da chave de criptografia em um local físico, exemplo em um CD, DVD, dispositivo USB.

Proteja a chave de criptografia com DPAPI:

```
string Protect(string chave) {  
    try {  
        byte[] segredo = Encoding.ASCII.GetBytes(chave);  
        byte[] entropia = { 10, 8, 4, 6, 5 };  
        byte[] b = ProtectedData.Protect(segredo, entropia, DataProtectionScope.LocalMachine);  
        return Convert.ToBase64String(b);  
    }  
    catch (CryptographicException) {  
        //  
    }  
    return null;  
}
```

Armazene a chave de criptografia protegida com DPAPI no registro do Windows:

```
[RegistryPermissionAttribute(SecurityAction.Demand,  
    Create = @"HKEY_LOCAL_MACHINE\Software\Microsoft\ChaveTeste")]  
void GravarNoRegistro(string valor) {  
    try {  
        RegistryKey rk = Registry.LocalMachine.OpenSubKey("Software\\microsoft", true);  
        RegistryKey chave = rk.CreateSubKey("ChaveTeste");  
        chave.SetValue("key", valor);  
    }  
    catch (UnauthorizedAccessException) {  
        //  
    }  
    catch (SecurityException) {  
        //  
    }  
    catch (Exception) {  
        //  
    }  
}
```

Proteja a chave do registro com as permissões a seguir:

- Administradores: controle total
- Conta de processo (AUTORIDADE NT\SERVIÇO DE REDE): somente leitura

Obs.: nos arquivos de exemplo do livro você encontra uma aplicação Windows que exemplifica o que foi abordado nesse tópico.

12.10.7.1 Criptografe as informações sigilosas

Obtenha as informações que você deseja criptografar. Por exemplo, de um campo de formulário.

Acesse a chave do registro (ChaveTeste) e recupere a chave de criptografia:

```
[RegistryPermissionAttribute(SecurityAction.Demand,
    Read = @"HKEY_LOCAL_MACHINE\Software\Microsoft\ChaveTeste")]
string ReadRegistro() {
    RegistryKey ch = Registry.LocalMachine.OpenSubKey("Software\\Microsoft\\ChaveTeste", false);
    return ch.GetValue("key") as string;
}
```

Use DPAPI para descriptografar a chave de criptografia extraída do registro:

```
string Unprotect(string segredo) {
    try {
        byte[] segredo = Convert.FromBase64String(segredo);
        byte[] entropia = { 10, 8, 4, 6, 5 };
        byte[] b = ProtectedData.Unprotect(segredo, entropia, DataProtectionScope.LocalMachine);
        return Convert.ToBase64String(b);
    }
    catch (CryptographicException) {
        //
    }
    return null;
}
```

Use a chave de criptografia e a classe `RijndaelManaged` para criptografar as informações sigilosas:

```
private string Criptografar(string dados, string chave) {
    byte[] b = Encoding.UTF8.GetBytes(dados);
    byte[] pw = Encoding.UTF8.GetBytes(chave);

    RijndaelManaged rm = new RijndaelManaged();

    PasswordDeriveBytes pdb = new PasswordDeriveBytes(chave,
        new MD5CryptoServiceProvider().ComputeHash(pw));
    rm.Key = pdb.GetBytes(32);
    rm.IV = pdb.GetBytes(16);
    rm.BlockSize = 128;
    rm.Padding = PaddingMode.PKCS7;

    MemoryStream ms = new MemoryStream();
```

```

    CryptoStream cryptStream = new CryptoStream(ms, rm.CreateEncryptor(rm.Key, rm.IV),
    CryptoStreamMode.Write);
    cryptStream.Write(b, 0, b.Length);
    cryptStream.FlushFinalBlock();
    return System.Convert.ToBase64String(ms.ToArray());
}

```

Exemplo:

```
string saida = Criptografar(txtDados.Text.Trim(), Unprotect(ReadRegistro()));
```

12.10.7.2 Descriptografe as informações sigilosas

Primeiramente, recupere as informações criptografadas, por exemplo, de um banco de dados.

Recupere e, em seguida, descriptografe com DPAPI a chave de criptografia armazenada no registro.

Use a classe `RijndaelManaged` para descriptografar as informações sigilosas:

```
string saida = Descriptografar(Convert.FromBase64String(CampoDoBD), Unprotect(ReadRegistro()));
```

12.11 Login da aplicação

A segurança da nossa aplicação de exemplo começa pela criação do banco de dados. A linha a seguir:

```
aspnet_regsql.exe -S .\SQLEXPRESS -E -d Northwind -A mr
```

Acrescenta as tabelas de segurança `aspnet_Membership`, `aspnet_Roles`, `aspnet_Users`, `aspnet_UsersInRoles`, `aspnet_Applications`, `aspnet_SchemaVersions` ao banco de dados Northwind.

Clique no menu **Iniciar > Todos os programas > Microsoft Visual Studio 2013 > Visual Studio Tools > Developer Command Prompt for VS2013**. Em seguida, digite:

```
aspnet_regsql.exe -S .\SQLEXPRESS -E -d Northwind -A mr
```

Modifique o arquivo de configuração `web.config` localizado na raiz da aplicação. Altere o elemento `connectionStrings`.

Substitua:

```
<add name="ApplicationServices" connectionString="data source=.\SQLEXPRESS;Integrated
Security=SSPI;AttachDBFilename=|DataDirectory|aspnetdb.mdf;User Instance=true"
providerName="System.Data.SqlClient" />
```

Por:

```
<add name="SqlBDseguro" connectionString="Data Source=.\SQLEXPRESS;
Integrated Security=SSPI;Initial Catalog=Northwind;" />
```

Altere o elemento `membership`.

Substitua:

```
<membership>
  <providers>
    <clear />
    <add name="AspNetSqlMembershipProvider" type="System.Web.Security.SqlMembershipProvider"
      connectionStringName="ApplicationServices" enablePasswordRetrieval="false"
      enablePasswordReset="true" requiresQuestionAndAnswer="false"
      requiresUniqueEmail="false" maxInvalidPasswordAttempts="5"
      minRequiredPasswordLength="6" minRequiredNonalphanumericCharacters="0"
      passwordAttemptWindow="10" applicationName="/" />
  </providers>
</membership>
```

Por:

```
<membership defaultProvider="SqlProvider" >
  <providers>
    <clear />
    <add
      name="SqlProvider"
      type="System.Web.Security.SqlMembershipProvider"
      connectionStringName="SqlBDseguro"
      applicationName="AppCapitulo12"
      enablePasswordRetrieval="false"
      enablePasswordReset="true"
      requiresQuestionAndAnswer="false"
      requiresUniqueEmail="true"
      minRequiredPasswordLength="8"
      minRequiredNonalphanumericCharacters="1"/>
    </providers>
  </membership>
```

O atributo `enablePasswordReset` quando definido como `true` permite que a aplicação recrie a senha no banco de dados. Recurso utilizado quando enviamos uma nova senha ao usuário.

O atributo `requiresQuestionAndAnswer` deve ser `false`, pois o formulário de cadastro-padrão adicionado à aplicação ASP.NET MVC não cadastra perguntas e respostas.

O atributo `applicationName` define o nome da aplicação e deve ser definido para evitar eventuais problemas de identificação ao migrar a aplicação de um servidor para outro. Ou seja, você precisa manter o mesmo nome ao migrar a aplicação de um servidor para outro ou você perde acesso aos usuários e regras de acesso previamente cadastrados.

Por fim, substitua:

```
<roleManager enabled="false">
  <providers>
    <clear />
    <add name="AspNetSqlRoleProvider" type="System.Web.Security.SqlRoleProvider"
      connectionStringName="ApplicationServices" applicationName="/" />
    <add name="AspNetWindowsTokenRoleProvider"
      type="System.Web.Security.WindowsTokenRoleProvider" applicationName="/" />
  </providers>
</roleManager>
```

Por:

```
<roleManager defaultProvider="SqlRoleProvider"
  enabled="true"
  cacheRolesInCookie="true"
  cookieName="{278FAD2B-26C2-4940-932B-4780E200569B}"
  cookieTimeout="5"
  cookieRequireSSL="false"
  cookieSlidingExpiration="true"
  cookieProtection="All"
  createPersistentCookie="false">
  <providers>
    <add
      name="SqlRoleProvider"
      type="System.Web.Security.SqlRoleProvider"
      connectionStringName="SqlBDseguro"
      applicationName="AppCapitulo12"/>
  </providers>
</roleManager>
```

Altere também o elemento forms. Substitua:

```
<authentication mode="Forms">
  <forms loginUrl="~/Account/LogOn" timeout="2880" />
</authentication>
```

Por:

```
<authentication mode="Forms">
  <forms loginUrl="~/Account/LogOn" timeout="2880"
    name="EF7B6C48-3DD6-45FD-871C-8374AE0D9F0D"
    protection="All"
    requireSSL="false"
    path="/"
    slidingExpiration="true" />
</authentication>
```

Obs.: não use o valor-padrão no atributo `cookieName` quando múltiplos websites estiverem hospedados no mesmo servidor. Nesse caso, defina um GUID como nome. clique no menu **Iniciar > Todos os programas > Microsoft Visual Studio 2010 > Microsoft Windows SDK Tools > GUID Generator**. Na janela **Create GUID**, gere um GUID no formato registro e copie para o atributo `cookieName`.

Acrescente ao banco de dados dois grupos de usuários (Clientes e Administradores) e, no mínimo, dois usuários. Siga os passos descritos no tópico “11.4 Gerenciando a aplicação”.

12.11.1 Formulário de login

A aplicação usa um formulário de login personalizado. Substitua o código do arquivo `Views/Account/LogOn.cshtml`:

```
@model AppCapitulo12.Models.LogOnModel
@{
    ViewBag.Title = "Tela de login";
}

<h2>Tela de login</h2>
<p>
    Por favor, digite o nome de usuário e senha. @Html.ActionLink("Clique aqui", "Register") se
    você não tem uma conta.
</p>

<script src="@Url.Content("~/Scripts/jquery.validate.min.js")" type="text/javascript"></script>
<script src="@Url.Content("~/Scripts/jquery.validate.unobtrusive.min.js")"
    type="text/javascript"></script>

@using (Html.BeginForm()) {
    @Html.AntiForgeryToken()
    @Html.ValidationSummary(true,
        "Login was unsuccessful. Please correct the errors and try again.")
    <div>
        <fieldset>
            <legend>Informações da conta</legend>

            <div class="editor-label">
                @Html.LabelFor(m => m.UserName)
            </div>
            <div class="editor-field">
                @Html.TextBoxFor(m => m.UserName)
                @Html.ValidationMessageFor(m => m.UserName)
            </div>
        </fieldset>
    </div>
}
```

```

<div class="editor-label">
    @Html.LabelFor(m => m.Password)
</div>
<div class="editor-field">
    @Html.PasswordFor(m => m.Password)
    @Html.ValidationMessageFor(m => m.Password)
</div>

<div class="editor-label">
    @Html.CheckBoxFor(m => m.RememberMe)
    @Html.LabelFor(m => m.RememberMe)
</div>

<p>
    <input type="submit" value="Enviar" />
</p>
</fieldset>
</div>
}

```

Pelo código:

```

@model AppCapítulo12.Models.LogOnModel
@{
    using (Html.BeginForm("LogOn", "Account")) {
        @Html.AntiForgeryToken()
        <table style="width: 100%; text-align: right">
            <tr>
                <td colspan="6" style="text-align: center; background-color: red; height: 0px;">
                    <div>
                        @Html.ValidationMessageFor(m => m.UserName, "",
                            new { style = "font-family: Arial, Helvetica, sans-serif;
                                font-weight: bold; color: white; font-size: 10px;" })
                        @Html.ValidationMessageFor(m => m.Password, "",
                            new { style = "font-family: Arial, Helvetica, sans-serif;
                                font-weight: bold; color: white; font-size: 10px;" })
                    </div>
                </td>
            </tr>
            <tr>
                <td colspan="6" style="text-align: center; background-color: #006486; height: 3px;">
                </td>
            </tr>
            <tr>
                <td width="45%">

```

```

        <td style="text-align: center; width: 10%;">
            <a href="@Href("~/Account/register")" class="menulogin">Cadastre-se </a>
        </td>
        <td style="width: 15%; text-align: right;">
            @Html.TextBoxFor(m => m.UserName,
                new { MaxLength = 50, style = "text-align: left; width: 95%;" })
        </td>
        <td style="width: 15%; text-align: right;">
            @Html.PasswordFor(m => m.Password,
                new { MaxLength = 15, style = "text-align: left; width: 95%;" })
        </td>
        <td style="text-align: left;">
            <input alt="Login" type="submit" value="Entrar"
                style="font-family: Arial, Helvetica, sans-serif;font-size: 10px;" />
        </td>
        <td>
            <a href="@Href("~/Account/NovaSenha")" class="menulogin">Esqueceu a senha?</a>
        </td>
    </tr>
</table>
    }
}

```

12.11.1.1 Métodos de controlador

Carregue o método de controlador LoadFormLogin na marca “Formulário de login aqui” dentro da página de layout Views\Shared_Layout.cshtml:

```
@{Html.RenderAction("LoadFormLogin", "Account");}
```

No controlador AccountController crie o método LoadFormLogin com o seguinte código:

```

[ChildActionOnly]
public ActionResult LoadFormLogin() {
    if (User.Identity.IsAuthenticated && (User.IsInRole("Clientes") || User.
        IsInRole("Administradores"))) {
        MembershipUser usuario = Membership.GetUser(User.Identity.Name);
        ViewBag.UserName = usuario.UserName;
        ViewBag.LastLoginDate = usuario.LastLoginDate;
        return PartialView("Logado");
    }
    else {
        return PartialView("LogOn");
    }
}

```

Verificamos se o usuário está logado e se pertence ao grupo Clientes ou Administradores:

```
if (User.Identity.IsAuthenticated && (User.IsInRole("Clientes") ||
    User.IsInRole("Administradores")))) {
```

Para usuários logados “pegamos” o nome do usuário e a data do último login:

```
MembershipUser usuario = Membership.GetUser(User.Identity.Name);
ViewBag.UserName = usuario.UserName;
ViewBag.LastLoginDate = usuario.LastLoginDate;
return PartialView("Logado");
```

Crie o partial view Views/Account/Logado.cshtml:

```
<table style="width: 100%; text-align: center">
  <tr>
    <td style="text-align: right; width: 40%;">
    </td>
    <td style="text-align: right; width: 30%;">
      <span style="font-family: Arial, Helvetica, sans-serif; font-weight: bold;
        color: white; font-size: 10px;">Último login:</span>
      <span style="font-family: Arial, Helvetica, sans-serif; color: white; font-size: 10px;">
        @ViewBag.LastLoginDate
      </span>
    </td>
    <td style="text-align: center; width: 73%;">
      <span style="font-family: Arial, Helvetica, sans-serif; font-weight: bold;
        color: white; font-size: 10px;">Seja bem vindo</span>
      <span style="font-family: Arial, Helvetica, sans-serif; color: white; font-size: 10px;">
        @ViewBag.UserName
      </span>
    </td>
    <td style="width: 7%; font-weight: bold; text-align: right;">
      <span><a href="@Href("~/Account/LogOff")" class="menulogin">Sair</a> </span>
    </td>
  </tr>
</table>
```

Acrescente o seletor CSS menulogin:

```
.menulogin:link {
  text-decoration: none;
  font-family: Arial, Helvetica, sans-serif;
  color: white;
  font-size: 10px;
}
.menulogin:visited {
  text-decoration: none;
  font-family: Arial, Helvetica, sans-serif;
```

```

        color: white;
        font-size: 10px;
    }
    .menulogin:active {
        text-decoration: none;
        font-family: Arial, Helvetica, sans-serif;
        color: white;
        font-size: 10px;
    }
    .menulogin:hover {
        text-decoration: underline;
        font-family: Arial, Helvetica, sans-serif;
        color: black;
        font-size: 10px;
    }
}

```

Para usuários anônimos exibimos o formulário de login:

```
return PartialView("LogOn");
```

O método de controlador `LogOn` autentica o usuário e grava o cookie de autenticação. Substitua o código original pelo código a seguir:

```

[HttpPost, ValidateAntiForgeryToken]//, RequireHttps
public ActionResult LogOn(LogOnModel model, string returnUrl) {
    if (ModelState.IsValid) {
        if (MembershipService.ValidateUser(model.UserName, model.Password)) {
            FormsService.SignIn(model.UserName, false);//model.RememberMe
            if (Url.IsLocalUrl(returnUrl)) {
                return Redirect(returnUrl);
            }
            else {
                return RedirectToAction("Index", "Home");
            }
        }
        else {
            return View("MensagemErro");
        }
    }
    return View(model);
}

```

No código incluímos alguns atributos de segurança:

```
[HttpPost, ValidateAntiForgeryToken]//, RequireHttps
```

Alteramos a linha:

```
FormsService.SignIn(model.UserName, false);//model.RememberMe
```

E acrescentamos o trecho de código a seguir:

```
else {
    return View("MensagemErro");
}
```

Crie o partial view `Views/Account/MensagemErro.cshtml` que contém a mensagem de erro exibida quando digitamos credenciais inexistentes:

```
<p style="font-family: Arial, Helvetica, sans-serif; font-size: 10px; font-weight: bold; color: Red;">
    Senha e/ou login incorretos. Por favor, digite novamente e clique em 'Entrar'
</p>
```

12.11.2 Enviando uma nova senha

Para enviar a senha para os usuários que esqueceram a senha usamos um formulário de campo único.

Ao informar o nome de usuário, o website cria e envia uma nova senha por e-mail.

Acrescente à aplicação o arquivo `Views/Account/NovaSenha.cshtml` com o código a seguir:

```
@model AppCapitulo12.Models.RegisterModel
@{
    ViewBag.Title = "Programando com ASP.NET MVC";
}
<div id="divProdutos" />
<table style="border-style: none; border-width: 0px; border-color: inherit; width: 100%; position: relative;">
    <tr>
        <td style="background-color: #027399; height: 0px;" colspan="3" class="secao">
            <strong>Digite o nome de usuário</strong>
        </td>
    </tr>
    <tr>
        <td style="vertical-align: top; text-align: left;">
            @{
                using (Html.BeginForm("NovaSenha", "Account")) {
                    @Html.AntiForgeryToken()
                    @Html.ValidationSummary(true);
                    <fieldset>
                        <table>
                            <tr>
                                <td style="vertical-align: top; height: 10px; text-align: left;">
                                </td>
                            </tr>
                        </table>
                    </fieldset>
                }
            }
        </td>
    </tr>
```

```

<td style="vertical-align: top; text-align: left;">
    <div class="editor-label">
        @Html.LabelFor(model => model.UserName)
    </div>
    <div class="editor-field">
        @Html.TextBoxFor(model => model.UserName,
            new { MaxLength = 50, @style = "width: 250px;" })
        @Html.ValidationMessageFor(model => model.UserName)
        <input type="submit" value="Enviar e-mail"
            style="font-family: Arial, Helvetica, sans-serif;
            font-size: 10px;" />
    </div>
</td>
</tr>
</table>
</fieldset>
}
}
</td>
</tr>
</table>

```

O resultado vemos na figura 12.22.

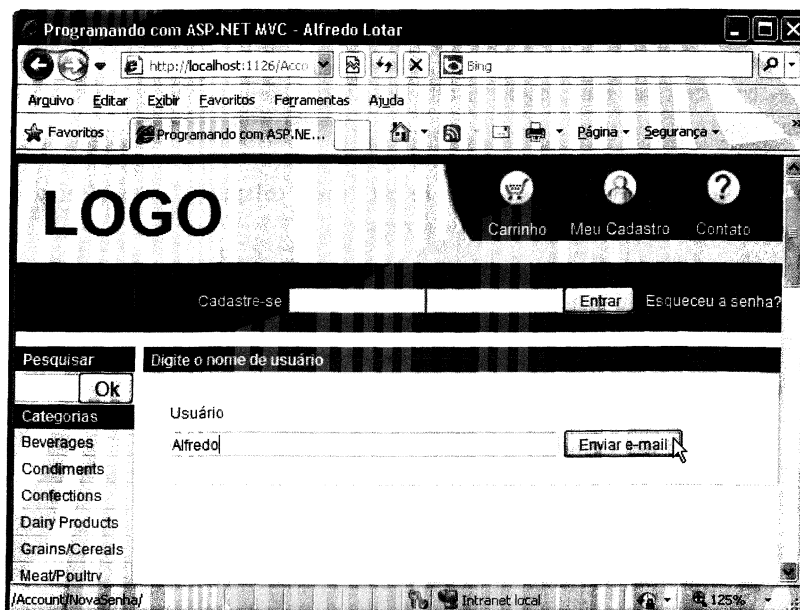


Figura 12.22 – Formulário de envio de nova senha.

12.11.2.1 Método de controlador

Repare no código anterior que no formulário invocamos o método de controlador NovaSenha:

```
using (Html.BeginForm("NovaSenha", "Account")) {
```

Crie esse método no controlador AccountController:

```
[HttpPost, ValidateAntiForgeryToken]
public ActionResult NovaSenha(string UserName) {
    MembershipUser membro = Membership.GetUser(UserName);
    if (membro != null) {
        string senha = membro.ResetPassword();
        SendEmail(membro.Email, "Nova senha", "Sua nova senha no website xx é:" + senha);
    }
    else {
        ModelState.AddModelError("", "Usuário não existe.");
    }
    return View();
}
```

Se o nome de usuário existe, a nova senha é criada e enviada via e-mail:

```
MembershipUser membro = Membership.GetUser(UserName);
if (membro != null) {
    string senha = membro.ResetPassword();
    SendEmail(membro.Email, "Nova senha", "Sua nova senha no website xx é:" + senha);
}
```

Pelo método SendEmail:

```
private void SendEmail(string email, string assunto, string mensagem) {
    try {
        WebMail.SmtpServer = "localhost";
        WebMail.SmtpPort = 25;
        //WebMail.EnableSsl = true;
        //WebMail.UserName = "usuário";
        //WebMail.Password = "senha";
        WebMail.From = "siteexemplo@siteexemplo.com";
        WebMail.SmtpUseDefaultCredentials = true;

        WebMail.Send(to: email, subject: assunto, body: mensagem);
        ModelState.AddModelError("", "Senha enviada com sucesso.");
    }
    catch (Exception ex) {
        ModelState.AddModelError("", "Erro ao enviar e-mail.");
    }
}
```

Acrescente o código do método `SendEmail` ao controlador `AccountController`.

Importe a namespace `System.Web.Helpers`:

```
using System.Web.Helpers;
```

12.12 Carrinho de compras

O carrinho de compras é constituído de itens. Cada item corresponde a um produto ou serviço.

Ao elaborar os requisitos do recurso carrinhos de compras verifique quais informações são essenciais, importantes e acessórias. Assim, concluímos que: precisamos de um carrinho de compras com itens.

Repare que temos uma relação do tipo um para muitos. Por exemplo, um carrinho de compras pode ter um ou mais itens.

Com base nessa informação, percebemos que necessitamos separar o carrinho de compras dos seus itens.

12.12.1 Criando as tabelas

Um website de venda de produtos necessita basicamente de duas tabelas para o carrinho de compras. Uma para armazenar o identificador do carrinho de compras e outra para os itens.

12.12.1.1 Tabela `ShoppingCart`

Então, quais informações o recurso carrinho de compras deve conter? Basicamente, três. O id do carrinho, a data de criação e um mecanismo de verificação para diferenciar os carrinhos abandonados dos que tiveram os pedidos finalizados. Você pode incluir outras informações se achar necessário.

Crie no banco de dados Northwind a tabela `ShoppingCart` com os campos a seguir:

Campo	Tipo de dados	Permite nulos	Especificação de identidade	Valor-padrão
<code>CartId</code>	<code>int</code>	Não	Sim	
<code>Data</code>	<code>datetime2(7)</code>	Não		<code>getdate()</code>
<code>IsCheckedOut</code>	<code>bit</code>	Não		0

Obs.: `getdate` é uma função do SQL Server.

12.12.1.2 Tabela ShoppingCartItems

A tabela ShoppingCartItems armazena os itens do carrinho de compras. E cada item contém um identificador de carrinho de compras, um identificador de produto, a quantidade de produtos e a data que o fato ocorreu, ou seja, quando o item foi adicionado ao carrinho.

Usando essas informações crie no banco de dados Northwind a tabela ShoppingCartItems:

Campo	Tipo de dados	Permite nulos	Especificação de identidade	Valor-padrão
CartItemId	int	Não	Sim	
CartId	int	Não		
ProductID	int	Não		
Data	datetime2(7)	Não		getdate()
Quantidade	smallint	Não		1

12.12.2 ADO.NET Entity Framework

Crie um diagrama no SQL Server com as tabelas ShoppingCart, ShoppingCartItems, Products do banco de dados Northwind. Observe a figura 12.23.

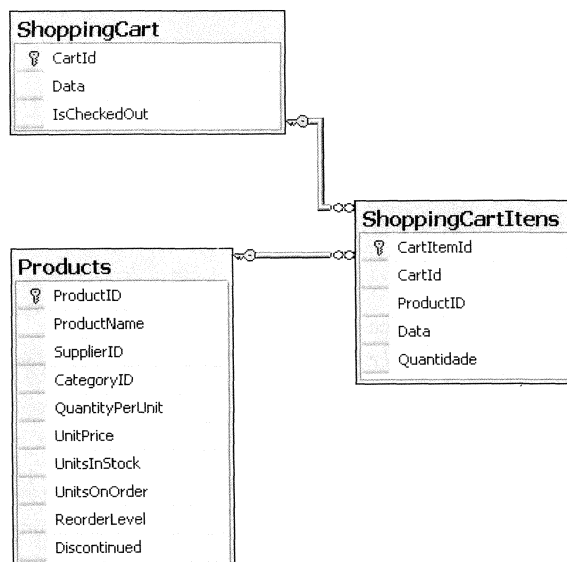


Figura 12.23 – Diagrama no sql server.

Atualize o modelo conceitual do ADO.NET Entity Framework. Abra o arquivo Models/Model1.edmx e clique com o botão direito do mouse em uma área livre do arquivo. Em seguida, selecione **Update Model from Database...**. Adicione as tabelas ShoppingCart e ShoppingCartItems.

12.12.3 Validação e formatação

Valide e formate as propriedades da classe `ShoppingCart`. Crie o arquivo `ShoppingCart.cs` no diretório `Models`. Acrescente o código a seguir ao arquivo:

```
using System.ComponentModel.DataAnnotations;
namespace AppCapitulo12.Models {
    [MetadataType(typeof(ShoppingCartMetadata))]
    public partial class ShoppingCart { }

    public class ShoppingCartMetadata {
        [Required(ErrorMessage = "O campo IsCheckedOut é obrigatório.")]
        [RegularExpression("^[true|false]+$", ErrorMessage = "Informe um valor booleano.")]
        public bool IsCheckedOut { get; set; }
    }
}
```

As propriedades `CartId` e `Data` não necessitam ser validadas. A entrada é realizada automaticamente pelo SQL Server.

Em seguida, valide as propriedades da classe `ShoppingCartItem` (é N no final mesmo). Adicione o arquivo `ShoppingCartItem.cs` ao diretório `Models` com o código a seguir:

```
using System.ComponentModel.DataAnnotations;
namespace AppCapitulo12.Models {
    [MetadataType(typeof(ShoppingCartItemMetadata))]
    public partial class ShoppingCartItem { }

    public class ShoppingCartItemMetadata {
        [Required(ErrorMessage = "O campo CartId é obrigatório.")]
        [RegularExpression("^[0-9]+$", ErrorMessage = "Informe um valor numérico.")]
        public int CartId { get; set; }

        [Required(ErrorMessage = "O campo ProductID é obrigatório.")]
        [RegularExpression("^[0-9]+$", ErrorMessage = "Informe um valor numérico.")]
        public int ProductID { get; set; }

        [Required(ErrorMessage = "O campo Quantidade é obrigatório.")]
        [RegularExpression("^[0-9]+$", ErrorMessage = "Informe um valor numérico.")]
        [Range(typeof(short), "1", "100", ErrorMessage = "Valor fora do intervalo. 1-100")]
        public short Quantidade { get; set; }
    }
}
```

12.12.4 Classe controladora

Para elaborar os métodos de controlador do carrinho de compras crie a classe de controlador `ShoppingController`. Em seguida, acrescente o método construtor:

```
private IGenericRepository _repository;
public ShoppingController() {
    _repository = new EntityFrameworkRepository(new NorthwindBD());
}
```

E também o código a seguir:

```
protected override void HandleUnknownAction(string actionName) {
    try {
        this.View(actionName).ExecuteResult(this.ControllerContext);
    }
    catch (InvalidOperationException) {
        this.View("NotFound").ExecuteResult(this.ControllerContext);
    }
}
```

Importe a namespace `AppCapitulo12.Models`:

```
using AppCapitulo12.Models;
```

12.12.4.1 Cookies

O identificador do carrinho de compras é gravado em um cookie. Quando acessamos o carrinho de compras, verificamos se o cookie existe e extraímos o identificador.

Para gravar e ler cookies crie o arquivo `Cookies.cs` no diretório `Models` e acrescente o código a seguir:

```
using System;
using System.Text.RegularExpressions;
using System.Web;
namespace AppCapitulo12.Models {
    public static class Cookies {
        public static int LerCookie() {
            int resultado;
            if (HttpContext.Current.Request.Cookies["ShoppingCartId"] != null)
                if (Regex.IsMatch(HttpContext.Current.Request
                    .Cookies["ShoppingCartId"].Value, "^[0-9]+$"))
                    if (Int32.TryParse(HttpContext.Current.Request
                        .Cookies["ShoppingCartId"].Value, out resultado))
                        return resultado;
            return 0;
        }
    }
}
```

```
public static bool CartIdExist() {  
    if (HttpContext.Current.Request.Cookies["ShoppingCartId"] != null)  
        return Regex.IsMatch(HttpContext.Current.Request  
            .Cookies["ShoppingCartId"].Value, "[0-9]+$");  
    return false;  
}  
  
public static void GravarCookie(string valor) {  
    var cook = new HttpCookie("ShoppingCartId");  
    cook.HttpOnly = true;  
    cook.Value = valor;  
    cook.Expires = DateTime.Now.AddYears(1);  
    HttpContext.Current.Response.Cookies.Add(cook);  
}  
}
```

12.12.4.2 Métodos de controlador

Para o carrinho de compras desenvolvemos três métodos de controlador. Um exibe os itens atuais do carrinho, outro adiciona novos itens e um terceiro método exclui os itens.

Para desenvolver o código do método de controlador `GetShoppingCartItems` observe o fluxograma da figura 12.24.



Figura 12.24 – Fluxograma do método `getshoppingcartitems`.

Observando o fluxograma percebemos que o método `GetShoppingCartItems` exibe uma mensagem específica se o carrinho de compras não existe ou é vazio:

```
if (!Cookies.CartIdExist || !CartHasItem())
    return View("CarrinhoVazio");
}
```

O partial view `CarrinhoVazio` exibe a mensagem ao usuário. Crie o arquivo `CarrinhoVazio.cshtml` no diretório `Views/Shopping` e inclua o código a seguir:

```
<div id="divProdutos" />
<p style="font-family: Arial, Helvetica, sans-serif; font-size: 10px; font-weight: bold;
    color: Red;">
    @ViewBag.Mensagem
</p>
```

Crie também o método `CartHasItem`. O método `CartHasItem` verifica com base no identificador do carrinho de compras se há itens armazenados no banco de dados:

```
private bool CartHasItem() {
    int cartId = Cookies.LerCookie();
    if (cartId == 0) return false;
    var model = _repository.GetSingle<ShoppingCartItem>(x => x.CartId == cartId);
    if (model == null) {
        return false;
    }
    else {
        return true;
    }
}
```

Os itens atuais do carrinho são exibidos com o auxílio do método `Find`:

```
int cartId = Cookies.LerCookie();
var model = _repository.Find<ShoppingCartItem>(x => x.CartId == cartId);
var modelo = _repository.Find<ShoppingCartItem>(x => x.CartId == cartId).Sum(s =>
    (s.Quantidade * s.Product.UnitPrice));
Session["Total"] = modelo.Value;
if (model == null)
    return View("NotFound");
else
    return View("GetShoppingCartItens", model);
```

O método `LerCookie` retorna o identificador do carrinho de compras:

```
int cartId = Cookies.LerCookie();
```

O total do carrinho é calculado com base em informações extraídas do banco de dados e armazenado em uma variável de sessão:

```
var modelo = _repository.Find<ShoppingCartItem>(x => x.CartId == cartId).Sum(s =>
    (s.Quantidade * s.Product.UnitPrice));
Session["Total"] = modelo.Value;
```

Adicione o método `GetShoppingCartItems` ao controlador `ShoppingController`:

```
public ActionResult GetShoppingCartItems() {
    ViewBag.Mensagem = "Seu carrinho de compras está vazio.";
    ViewBag.Categoria = "Carrinho de compras";
    if (!Cookies.CartIdExist() || !CartHasItem()) {
        return View("CarrinhoVazio");
    }
    else {
        try {
            int cartId = Cookies.LerCookie();
            var model = _repository.Find<ShoppingCartItem>(x => x.CartId == cartId);
            var modelo = _repository.Find<ShoppingCartItem>(x =>
                x.CartId == cartId).Sum(s => (s.Quantidade * s.Product.UnitPrice));
            Session["Total"] = modelo.Value;
            if (model == null)
                return View("NotFound");
            else
                return View("GetShoppingCartItems", model);
        }
        finally {
            if (_repository != null)_repository.Dispose();
        }
    }
}
```

12.12.4.2.1 Adicionando itens ao carrinho

Para acrescentar um novo item ao carrinho de compras usamos o método de controlador `AddShoppingCartItem`:

```
public ActionResult AddShoppingCartItem([Bind(Exclude = "IsCheckedOut")]int ProductID) {
    try {
        if (!Cookies.CartIdExist()) {
            CreateShoppingCart();
            AddCartItems(ProductID);
        }
        else {
            AddCartItems(ProductID);
        }
    }
    catch (System.Data.EntitySqlException) {
        //
    }
    catch (System.Data.EntityCommandExecutionException) {
        //
    }
}
```



```
finally {  
    if (_repository != null) _repository.Dispose();  
}  
return RedirectToAction("GetShoppingCartItems");  
}
```

Observe a figura 12.25.

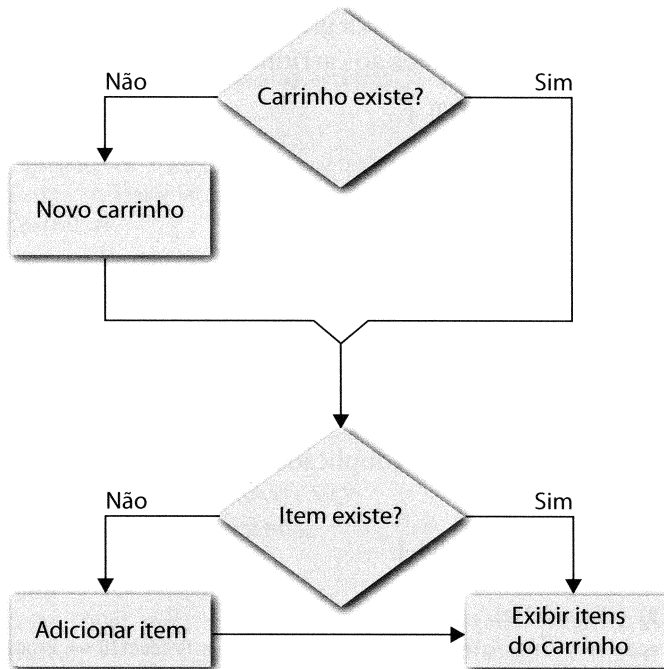


Figura 12.25 – Fluxograma do método `addshoppingcartitem`.

Quando observamos o código e o fluxograma, notamos que, primeiramente, verificamos se o carrinho de compras existe.

```
if (!Cookies.CartIdExist()) {
```

Se o carrinho de compras existe, simplesmente acrescentamos o novo item:

```
AddCartItems(ProductID);
```

Caso contrário, criamos o carrinho de compras antes de acrescentar os itens:

```
CreateShoppingCart();
```

```
AddCartItems(ProductID);
```

O método `CreateShoppingCart` adiciona o carrinho de compras ao banco de dados e, em seguida, retorna o identificador e o armazena em um cookie:

```
private void CreateShoppingCart() {
    ShoppingCart entidade = new ShoppingCart();
    entidade.Data = DateTime.Now;
    _repository.Create<ShoppingCart>(entidade);
    var model = _repository.GetAll<ShoppingCart>().Max(x => x.CartId);
    if (model > 0) Cookies.GravarCookie(model.ToString());
}
```

O método `AddCartItem` extrai o identificador gravado pelo método `CreateShoppingCart` em um cookie e acrescenta os novos itens ao carrinho de compras:

```
private void AddCartItem(int ProductID) {
    if (!CartItemExist(ProductID)) {
        ShoppingCartItem entidade = new ShoppingCartItem();
        entidade.CartId = Cookies.LerCookie();
        entidade.ProductID = ProductID;
        entidade.Data = DateTime.Now;
        entidade.Quantidade = 1;
        _repository.Create<ShoppingCartItem>(entidade);
    }
}
```

O método `CartItemExist` evita que itens duplicados sejam adicionados ao carrinho de compras:

```
private bool CartItemExist(int ProductID) {
    int cartId = Cookies.LerCookie();
    if (cartId == 0) return false;
    var model = _repository.GetSingle<ShoppingCartItem>(x => x.ProductID == ProductID &&
        x.CartId == cartId);
    if (model == null) {
        return false;
    }
    else {
        return true;
    }
}
```

12.12.4.2.2 Excluindo itens do carrinho

Excluimos itens do carrinho de compras com o método `DelShoppingCartItem`:

```
public ActionResult DelShoppingCartItem(int ProductID) {
    try {
        ShoppingCartItem entidade = new ShoppingCartItem();
        _repository.Delete<ShoppingCartItem>(x => x.ProductID == ProductID);
    }
}
```

```

        catch (System.Data.EntitySqlException) {
            //
        }
        catch (System.Data.EntityCommandExecutionException) {
            //
        }
        finally {
            if (_repository != null) _repository.Dispose();
        }
        return RedirectToAction("GetShoppingCartItems");
    }
}

```

Obs.: acrescente os métodos `AddShoppingCartItem`, `CreateShoppingCart`, `AddCartItems`, `CartItemExist`, `DelShoppingCartItem` e seu respectivo código ao controlador `ShoppingController`.

12.12.4.2.3 Carrinho do compras

Acrescente ao diretório `Views/Shopping` o arquivo `GetShoppingCartItems.cshtml` com o código a seguir:

```

@model IEnumerable<AppCapitulo12.Models.ShoppingCartItem>
@{
    ViewBag.Title = "Programando com ASP.NET MVC";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
<div id="divProdutos" />
<table style="border-style: none; border-width: 0px; border-color: inherit; width: 100%;
position: relative;">
    <tr>
        <td style="background-color: #027399; height: 0px;" colspan="3" class="secao">
            <strong>@ViewBag.Categoria</strong>
        </td>
    </tr>
    <tr>
        <td style="vertical-align: top; text-align: left;">
            <table class="grade" runat="server">
                <tr>
                    <td class="header">
                        Produtos
                    </td>
                    <td class="header">
                        Quantidade
                    </td>
                    <td class="header">
                        Valor unitário
                    </td>
                    <td class="header">
                        Valor total

```

```

        </td>
        <td class="header">
            Excluir
        </td>
    </tr>
    @foreach (var item in Model) {
        <tr>
            <td class="letras" style="width: 50%; text-align: left;">
                @item.Product.ProductName
            </td>
            <td class="letras" style="width: 15%; text-align: left;">
                @item.Quantidade
            </td>
            <td class="letras" style="width: 20%; text-align: left;">
                @String.Format(new System.Globalization.CultureInfo("pt-BR"),
                    "{0:c}", item.Product.UnitPrice)
            </td>
            <td class="letras" style="width: 20%; text-align: left;">
                @String.Format(new System.Globalization.CultureInfo("pt-BR"),
                    "{0:c}", (item.Quantidade * item.Product.UnitPrice))
            </td>
            <td class="letras" style="width: 20%; text-align: left;">
                <a href="@Href("~/shopping/De1ShoppingCartItem?ProductID=")@item.ProductID">
                    </a>
                </td>
        </tr>
        <tr>
            <td style="background-image:url('@Href("~/Content/Imagens")/barra.gif');
                height:3px;" colspan="5">
            </td>
        </tr>
    }
    <tr>
        <td colspan="2">
            </td>
        <td class="letras" style="text-align: right;">
            <strong>Total:</strong>
        </td>
        <td class="letras" style="text-align: left;">
            <strong>@String.Format(new System.Globalization.CultureInfo("pt-BR"),
                "{0:c}", Session["Total"])</strong>
        </td>
    </tr>
</table>
</td>
</tr>
</table>

```

12.12.5 Adicione outros recursos

Desenvolvemos a estrutura básica da aplicação de exemplo. Você pode adicionar outros recursos. Sugerimos que você acrescente os seguintes recursos:

- Modifique o mecanismo de busca. Permita a busca de múltiplos termos etc. Use outros mecanismos de busca como referência.
- Altere o design da aplicação.
- Adicione ao banco de dados Northwind a tabela Pedidos:

Campo	Tipo de dados	Permite nulos	Especificação de identidade	Valor padrão
IdPedido	int	Não	Sim	
CardId	int	Não		
IdCadastro	int	Não		
DataPedido	datetime2(7)	Não		getdate()
EnderecoEntrega	ntext	Não		
TotalDoPedido	money	Não		

Uma aplicação profissional necessita armazenar diversas outras informações, como: IdStatusPedido, DataPagamento, DataEntrega, ValorFrete, TipoPagamento, TipoEntrega.

- Crie a tabela PedidosItens no banco de dados Northwind:

Campo	Tipo de dados	Permite nulos	Especificação de identidade
IdPedidoItens	int	Não	Sim
IdPedido	int	Não	
ProductID	int	Não	
Quantidade	smallint	Não	
PrecoUnitario	money	Não	

- No carrinho de compras acrescente o botão “finalizar pedido” e desenvolva os métodos de controlador necessários para gravar o pedido do usuário no banco de dados.
- Adicione também o recurso de atualização da quantidade de itens do carrinho de compras.
- Crie uma área administrativa para o website. Permita excluir, atualizar informações de clientes.
- Inclua recursos necessários para adicionar, excluir, atualizar produtos e categorias.
- Crie também um mecanismo para gerenciar os pedidos recebidos.
- Inclua o cálculo de fretes no carrinho de compras.

12.12.6 Publicando a aplicação

Antes de enviar a aplicação para o servidor web da empresa de web hosting, é necessário preparar os arquivos do seu website. Siga os seguintes passos:

- Certifique-se de que a aplicação foi realmente finalizada.
- Configure os elementos authentication, compilation, trace, customErrors, httpCookies, sessionState, pages, globalization e roleManager no arquivo de configuração web.config da seguinte forma:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.web>
    <authentication mode="Forms">
      <forms name="{3A488A0C-76AE-457B-8CB9-4DD402017860}" loginUrl="/Login.aspx">
        </forms>
      </authentication>
    <compilation debug="false" targetFramework="4.0"/>
    <trace enabled="false"/>
    <customErrors mode="RemoteOnly"/>
    <httpCookies httpOnlyCookies="true"/>
    <sessionState cookieless="false" />
    <globalization fileEncoding="iso-8859-1" requestEncoding="iso-8859-1"
      responseEncoding="iso-8859-1" culture="pt-BR" uiCulture="pt-BR"/>

    <roleManager defaultProvider="SqlRoleProvider"
      enabled="true"
      cacheRolesInCookie="true"
      cookieName="{278FAD2B-26C2-4940-932B-4780E200569B}"
      cookieTimeout="5"
      cookieRequireSSL="true"
      cookieSlidingExpiration="true"
      cookieProtection="All"
      createPersistentCookie="false">
    </roleManager>
  </system.web>
</configuration>
```

- Envie para a empresa de web hosting somente os arquivos utilizados pelo website, que geralmente têm alguma das seguintes extensões: .aspx, .ascx, .cshtml, .css, .skin, .js, .asax, .master, .dll etc. Não envie arquivos .cs ou .vb, pois qualquer pessoa que tenha acesso físico ao servidor poderá ver o código-fonte da sua aplicação web.

- Se estiver tudo certo, envie os arquivos para a empresa de web hosting via FTP. A senha, o nome de usuário e outras configurações usadas pelo software-cliente FTP são fornecidas pela empresa de web hosting.
- Recrie nos servidores da empresa de web hosting a mesma estrutura de diretórios que o website tem no seu computador. Exemplo: DLLs no diretório Bin, skins no diretório App_Themes e assim por diante.
- Depois de enviar os arquivos, teste todos os hyperlink do website.
- Se tiver algum problema técnico, utilize o suporte da empresa de web hosting.

Antes de enviar os arquivos, é interessante você compilar as classes da aplicação em DLLs separadas. Por exemplo, o Visual Studio cria uma única DLL:

```
C:\Livro\AppCapítulo12\bin\AppCapítulo12.dll
```

Uma boa prática é dividir a aplicação em várias DLLs. Coloque as classes do diretório Models em uma DLL, as classes do diretório Controllers em outra.

Crie as DLLs manualmente. No **Developer Command Prompt for VS2013**, digite, por exemplo:

```
csc /r:System.ComponentModel.DataAnnotations.dll /r:System.Web.dll  
/r:System.Data.Entity.dll /t:library /out:Model.dll Product.cs Cadastro.cs Category.cs  
Contato.cs Cookies.cs ShoppingCartItem.cs ShoppingCart.cs IGenericRepository.cs  
EntityFrameworkRepository.cs
```

E crie o arquivo Model.dll. Após criar as DLLs da aplicação, coloque-as no diretório Bin.

12.12.7 Unidade de teste

As unidades de teste permitem que você teste métodos de uma classe. Por exemplo, é possível fazer comparações e verificar se um método retorna verdadeiro, falso, nulo.

Quando criamos um novo projeto ASP.NET MVC, temos a opção de adicionar uma unidade de testes.

Observe a figura 12.26.

Quando clicamos em **OK**, é acrescentado o seguinte código:

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Web.Mvc;  
using Microsoft.VisualStudio.TestTools.UnitTesting;  
using MvcApplication3;  
using MvcApplication3.Controllers;  
  
namespace MvcApplication3.Tests.Controllers {
```

```

[TestClass]
public class HomeControllerTest {
    [TestMethod]
    public void Index() {
        // Arrange
        HomeController controller = new HomeController();
        // Act
        ViewResult result = controller.Index() as ViewResult;

        // Assert
        Assert.AreEqual("Welcome to ASP.NET MVC!", result.ViewBag.Message);
    }
    [TestMethod]
    public void About() {
        // Arrange
        HomeController controller = new HomeController();

        // Act
        ViewResult result = controller.About() as ViewResult;

        // Assert
        Assert.IsNotNull(result);
    }
}
}

```

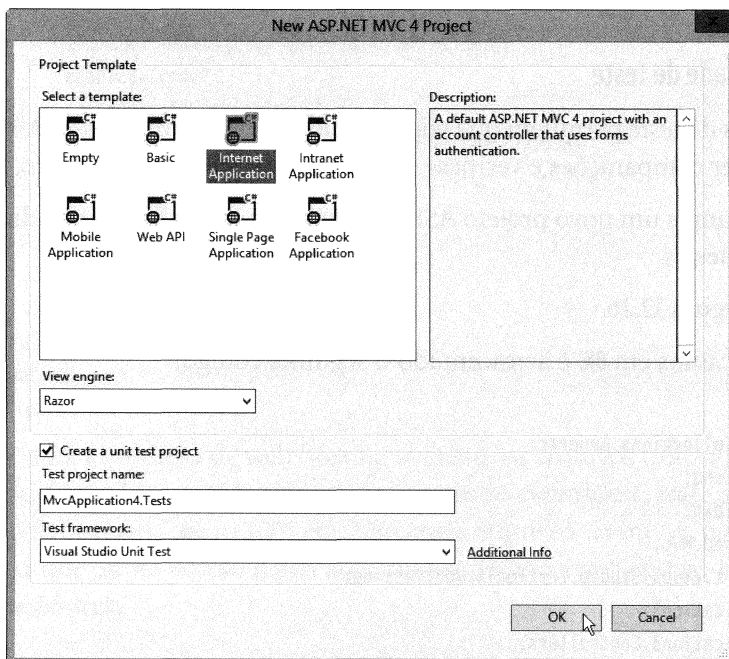


Figura 12.26 – Adicionando uma unidade de testes.

Se você tem um projeto e quer adicionar uma unidade de testes, use a barra de ferramentas Test Tools. Observe a figura 12.27.

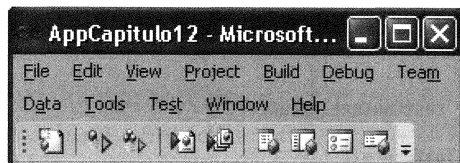


Figura 12.27 – Barra de ferramentas test tools.

As opções na barra de ferramentas nos permitem criar um nova unidade de testes, executar testes no contexto atual ou na solução atual, e é claro exibir os resultados dos testes.

Quando clicamos no primeiro botão à esquerda na barra de ferramentas Test Tools, acionamos a caixa de diálogo **Add New Test**. Observe a figura 12.28.

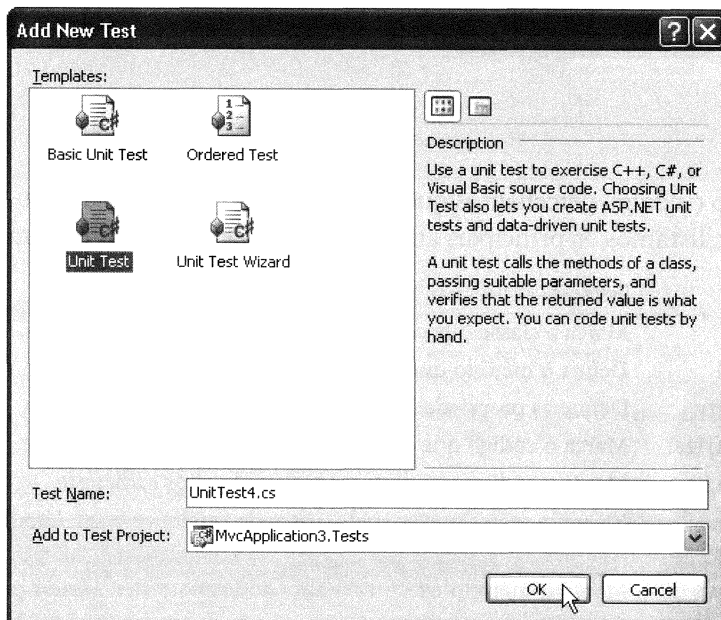


Figura 12.28 – Caixa de diálogo add new test.

Ao clicar em **OK** geramos o código a seguir:

```
using System;  
using System.Text;  
using System.Collections.Generic;  
using System.Linq;  
using Microsoft.VisualStudio.TestTools.UnitTesting;  
  
namespace MvcApplication3.Tests {
```

```

[TestClass]
public class UnitTest4 {
    public UnitTest4() {
        // TODO: Add constructor logic here
    }

    private TestContext testContextInstance;

    public TestContext TestContext {
        get {
            return testContextInstance;
        }
        set {
            testContextInstance = value;
        }
    }

    [TestMethod]
    public void TestMethod1() {
        // TODO: Add test logic here
    }
}

```

Observando o código percebemos que a classe e o método são marcados por atributos. A seguir, listamos os principais atributos que você pode usar para realizar testes.

Atributo	Descrição
TestClass	Marca a classe que contém os métodos que serão testados.
TestMethod	Define o método que será testado.
TestProperty	Define as propriedades que serão testadas em um método.
TestInitialize	Marca o código que deve ser executado antes de cada teste.
TestCleanup	Marca o código que deve ser executado após cada teste.
ClassInitialize	Marca o método executado antes de qualquer teste. Usado para alocar recursos.
ClassCleanup	Marca o método executado após todos os testes. Usado para liberar os recursos.
Ignore	Desativa os testes no método marcado. Útil quando você deseja desativar temporariamente os testes em determinado método.

12.12.7.1 Executando os testes

No Visual Studio os testes são realizados com o auxílio de três classes: `Assert`, `CollectionAssert`, `StringAssert`.

A classe `Assert` faz comparações com informações manipuladas ou retornadas por um método. Por exemplo, `IsNotNull` verifica valores nulos.

A classe `CollectionAssert` verifica elementos em coleções. Por exemplo, `AllItemsAreUnique` testa se a coleção contém ou não informações exclusivas.

Para testar strings temos a classe `StringAssert`. Por exemplo, `Matches` e `DoesNotMatch` realizam verificações por meio de expressões regulares.

Para realizar os testes, geralmente você precisa criar uma instância da classe de controlador que contém os métodos-alvo do teste. Exemplo:

```
public void Index() {
    HomeController controller = new HomeController();
    ViewResult result = controller.Index() as ViewResult;
    Assert.AreEqual("Welcome to ASP.NET MVC!", result.ViewBag.Message);
}
```

Após a definição do código, use a barra de ferramentas `Test Tools` para realizar os testes na aplicação. O resultado vemos na figura 12.29.

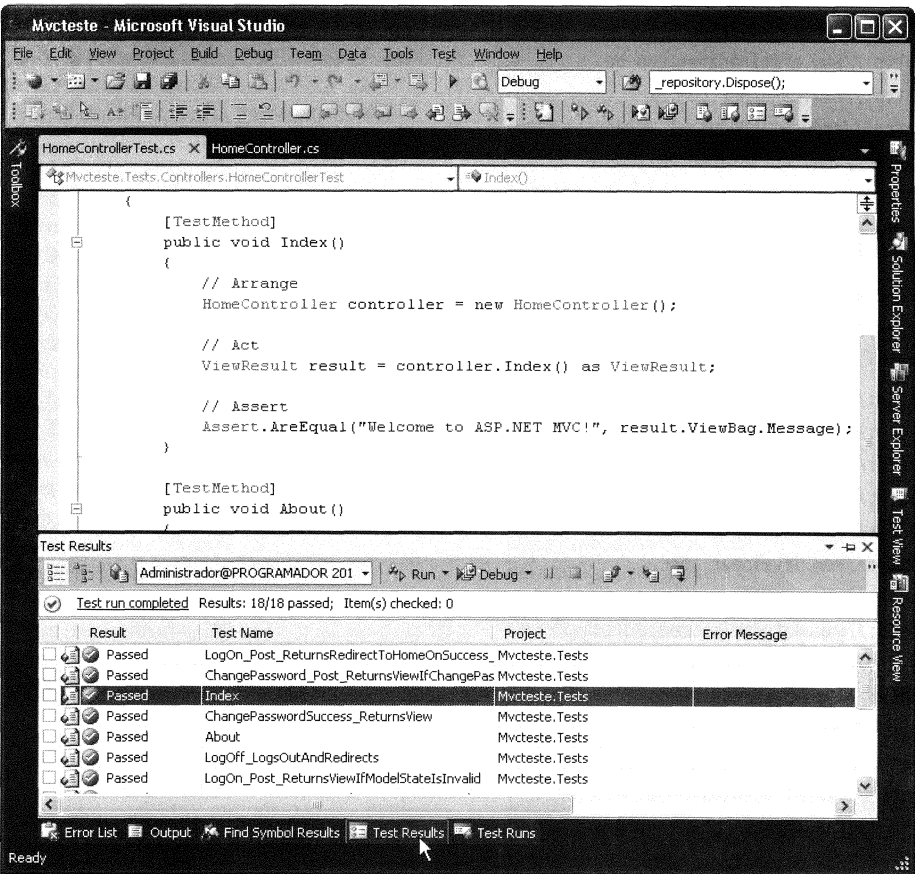


Figura 12.29 – Resultado dos testes.

Sites ASP.NET em português

<http://www.msdnbrasil.com.br>
<http://www.csharpbrasil.com.br>
<http://www.linhadecodigo.com.br>
<http://www.imasters.com.br>
<http://www.macoratti.net>
<http://www.bufaloinfo.com.br>
<http://www.oficinadanet.com.br>
<http://www.csharpbr.com.br>
<http://programandodotnet.wordpress.com>
<http://www.fabriciosanchez.com.br>

Sites ASP.NET em inglês

<http://www.asp.net>
<http://msdn.microsoft.com/asp.net>
<http://www.codeproject.com>
<http://www.eggheadcafe.com>
<http://www.aspfree.com>
<http://www.aspalliance.com>
<http://www.123aspx.com>
<http://www.411asp.net>
<http://www.developer.com>
<http://www.experts-exchange.com>
<http://www.dotnetbips.com>
<http://www.c-sharpcorner.com>
<http://www.mastercsharp.com>
<http://www.codeguru.com>
<http://windowsclient.net>
<http://www.pinvoke.net>
<http://regexlib.com>
<http://www.dotnetfunda.com>

Símbolos

@ Page, diretivas, 42, 85

A

AcceptVerbs, atributos, 51
 ActionFilterAttribute, classes, 258
 ActionLink, métodos, 37, 183, 263
 ActionMethodSelectorAttribute, classes, 57
 Action, métodos, 104, 186
 ActionName, atributos, 53
 ActionResult, classes, 48, 60
 AddCssClass, métodos, 225
 Add, elementos, 279
 AddImageWatermark, métodos, 166
 additionalHeaders, parâmetros, 177
 AddLegend, métodos, 167
 AddModelError, métodos, 174, 248
 AddSeries, métodos, 167
 AddTitle, métodos, 167
 AjaxOptions, classes, 265
 ajaxUpdateContainerId, parâmetros, 165
 AllItemsAreUnique, métodos, 387
 AllowHtml, atributos, 294
 alternatingRowStyle, parâmetros, 159
 AntiForgeryToken, métodos, 292
 App_Browsers, diretórios, 23
 App_Data, diretórios, 22
 App_GlobalResources, diretórios, 23
 applicationName, atributos, 360
 Application_Start, eventos, 181
 App_LocalResources, diretórios, 23
 App_Themes, diretórios, 23
 Areas, diretórios, 23
 aspnet_regsql, ferramentas, 278
 Assert, classes, 386
 AsyncController, classes, 78
 AsyncTimeout, atributos, 79
 AttributeEncode, métodos, 214
 Authorize, atributos, 291
 AuthorizeAttribute, classes, 260
 autoSortAndPage, parâmetros, 157

B

BeginForm, métodos, 176, 197, 267
 BeginRouteForm, métodos, 197
 Bind
 atributos, 54
 métodos, 157

C

CacheProfile, parâmetros, 74
 canPage, parâmetros, 157
 canSort, parâmetros, 157
 CAPTCHAs, 292
 caption, parâmetros, 160
 catch, instruções, 144
 Chart, 167
 chartType, parâmetros, 167
 CheckBox, métodos, 203
 ChildActionOnly, atributos, 62
 ClassCleanup, atributos, 386
 ClassInitialize, atributos, 386
 CollectionAssert, classes, 387
 Column, métodos, 158
 columnName, parâmetros, 158
 columnNames, parâmetros, 156
 Columns, métodos, 158
 Comentários, 141
 Compare, atributos, 234
 Content
 Diretórios, 22
 Métodos, 63
 ContentResult, classes, 63
 controlador, 44
 Controladores, 24
 Controllers, diretórios, 22
 Conversões, 145
 Crypto, 177
 CurrentCulture, propriedades, 253
 CurrentUICulture, propriedades, 253
 customErrors, elementos, 56, 297

D

DataBindTable, métodos, 169
 DataType, atributos, 233
 defaultSort, parâmetros, 156
 DefaultValue, atributos, 51
 Display, atributos, 233
 DisplayFormat, atributos, 233
 displayHeader, parâmetros, 160
 DisplayTextFor, métodos, 213
 Dispose, métodos, 125
 DoesNotMatch, métodos, 387
 DropDownListFor, métodos, 208
 DropDownList, métodos, 206
 Duration, parâmetros, 71

E

EditorFor, métodos, 213
 EditorForModel, métodos, 211
 Editor, métodos, 213
 EmptyResult, classes, 65
 emptyRowCellValue, parâmetros, 160
 EnableClientValidation, métodos, 236
 enablePasswordReset, atributos, 360
 EnableSsl, propriedades, 173
 EnableUnobtrusiveJavaScript, métodos, 236
 Encode, métodos, 214
 EndForm, métodos, 197
 ErrorMessageResourceName, propriedades, 253
 ErrorMessageResourceType, propriedades, 253
 ExceptionType, propriedades, 56
 Exclude, propriedades, 55

F

File, métodos, 176
 FileResult, classes, 64
 fillEmptyRows, parâmetros, 160
 FilterAttribute, classes, 256
 foreach, instruções, 143
 for, instruções, 143
 format, parâmetros, 158
 FormCollection, classes, 50
 forms, elementos, 70, 361

G

GenerateId, métodos, 225
 GetFromCache, métodos, 171
 GetHtml, métodos, 157, 158
 GetSelectLink, métodos, 162
 GetVaryByCustomString, métodos, 73
 global.asax, 180, 185
 Global.asax, 69, 73, 261
 Global.asax, métodos, 26

H

HandleError, atributos, 56
 HandleUnknownAction, métodos, 55
 head, elementos, 29
 helper, instruções, 224
 HiddenFor, métodos, 203
 Hidden, métodos, 203
 Href, métodos, 146
 htmlAttributes, parâmetros, 160

HtmlTextWriter, classes, 228
 HttpNotFound, métodos, 69
 HttpNotFoundResult, classes, 69
 HttpStatusCodeResult, classes, 69
 HttpUnauthorizedResult, classes, 69

I

IActionFilter, interface, 257
 IAuthorizationFilter, interface, 256
 IClientValidatable, interface, 245
 IExceptionFilter, interface, 257
 if, instruções, 141
 Ignore, atributos, 386
 Include, propriedades, 55
 inline, expressões, 36
 Insert, métodos, 76
 IsNull, métodos, 386
 IsValidForRequest, métodos, 57
 IsValid, métodos, 244

J

JavaScriptResult, classes, 66
 JsonResult, classes, 66

L

LabelFor, métodos, 199
 LabelForModel, métodos, 199
 Label, métodos, 198
 Layout, propriedades, 149, 153
 ListBox, métodos, 209, 211
 LoadingElementDuration, propriedades, 271
 LoadingElementId, propriedades, 271
 Location, parâmetros, 74

M

makecert, ferramentas, 298
 MapRoute, métodos, 180
 MasterPageFile, atributos, 33
 Master pages, 28
 Matches, métodos, 387
 membership, elementos, 279
 MergeAttribute, métodos, 225
 Message, propriedades, 297
 minutesToCache, parâmetros, 171
 Model Binders, 48
 model, palavras-chave, 146
 Models, 38

Models, diretórios, 22
MvcHtmlString, classes, 223

N

NoAsyncTimeout, atributos, 79
NonAction, atributos, 54
NoStore, parâmetros, 74
numericLinksCount, parâmetros, 161

O

ObjectContext, classes, 325
OnActionExecuted, métodos, 257
OnActionExecuting, métodos, 257
OnResultExecuted, métodos, 257
OnResultExecuting, métodos, 257
OnSuccess, propriedades, 270
Operadores, 145
Order, parâmetros, 261
OutputCache, atributos, 70
OutstandingOperation, propriedades, 79

P

Page, propriedades, 155
Partial, métodos, 103
partial, palavras-chave, 330
PartialView, métodos, 62
PasswordFor, métodos, 203
Password, métodos, 203
passwordStrengthRegularExpression, atributos, 290
Prefix, propriedades, 55
PrincipalPermission, atributos, 291

R

RadioButton, métodos, 205
Range, atributos, 234
Raw, métodos, 215
Razor, 139
Redirect, métodos, 186
RedirectPermanent, métodos, 186
RedirectResult, classes, 68
RedirectToAction, métodos, 68
RedirectToActionPermanent, métodos, 186
RedirectToRoutePermanent, métodos, 186
RedirectToRouteResult, classes, 68
RegisterRoutes, métodos, 26
RegularExpression, atributos, 234, 288

Remote, atributos, 235
Remove, métodos, 78
RenderAction, métodos, 104
RenderBody, métodos, 149, 153, 320
RenderPage, métodos, 151, 153
RenderPartial, métodos, 103, 106
RenderSection, método, 154
Require, atributos, 232
RequireHttps, atributos, 298
requiresQuestionAndAnswer, atributos, 360
RijndaelManaged, classes, 358
RNGCryptoServiceProvider, classes, 356
roleManager, elementos, 279
RouteLink, métodos, 184
RouteUrl, métodos, 187
rowsPerPage, parâmetros, 157

S

Save, métodos, 170
SaveToCache, métodos, 171
Scripts, diretórios, 22
SecureConnectionConstraint, classes, 182
Send, métodos, 173, 177
SessionID, propriedades, 58
Session_OnStart, eventos, 59
SessionState, atributos, 58
SessionStateBehavior, enumerações, 58
SHA256, métodos, 178
slidingExpiration, parâmetros, 171
SmtpPort, propriedades, 173
SmtpServer, propriedades, 172
source, parâmetros, 156
SqlDataReader, classes, 109
SqlDependency, parâmetros, 75
SSL, 298
StringAssert, classes, 387
StringLength, atributos, 233
Submit, métodos, 223
switch, instruções, 142
System.ComponentModel.DataAnnotations, namespaces, 231

T

TagBuilder, classes, 225
target, propriedades, 184
TempData, propriedades, 84
TestClass, atributos, 386
TestCleanup, atributos, 386
TestInitialize, atributos, 386
TestMethod, atributos, 386

TestProperty, atributos, 386
TextAreaFor, métodos, 202
TextArea, métodos, 201
TextBoxFor, métodos, 201
TextBox, métodos, 200
theme, parâmetros, 169
ToString, métodos, 225
try, instruções, 144
TryParse, métodos, 295
TryUpdateModel, métodos, 50

U

UrlHelper, classes, 226
using, palavras-chave, 224

V

ValidateAntiForgeryToken, atributos, 293, 342
ValidateInput, atributos, 294
ValidationAttribute, classes, 244
ValidationMessageFor, métodos, 240
ValidationMessage, métodos, 239
ValidationSummary, métodos, 237
VaryByContentEncoding, parâmetros, 73
VaryByHeader, parâmetros, 72
VaryByParam, parâmetros, 72
View
 métodos, 61
 propriedades, 56
ViewBag, propriedades, 35
ViewData, propriedades, 84
ViewResult, classes, 60
Views, diretórios, 22

W

web.config, 74, 236, 278
WebGrid, 155
WebImage, 165
WebMail, 172
where, cláusulas, 116
WriteFromCache, métodos, 171
Write, métodos, 165, 167

X

xValue, parâmetros, 167

Y

yValues, parâmetros, 167

Este livro apresenta muitos exemplos, códigos, dicas e conceitos relacionados ao ASP.NET MVC. No capítulo final o livro mostra passo a passo como desenvolver um website com ASP.NET MVC.

Aborda também os principais tópicos relacionados às versões 1, 2 e também os recursos acrescentados a versão 3, que se aplicam também às versões 4 e 5, como a sintaxe Razor, a propriedade ViewBag, os atributos Remote, Compare, AllowHtml, os novos tipos ActionResult, entre outros.

O ASP.NET MVC fornece por meio de design patterns uma maneira poderosa e alternativa para criar websites ASP.NET dinâmicos.

PROGRAMANDO COM

ASP.NET MVC

A seguir listamos os principais recursos abordados:

Sintaxe Razor
Os tipos retornados de ActionResult
Controladores Assíncronos
Exemplifica o uso de dezenas de atributos
Data Annotations Extensions
Cache
Variáveis de sessão
Criação de View e partial View
ADO.NET Entity Framework
Generic Repository
WebGrid Helper
Chart Helper
WebImage Helper
WebMail Helper
Crypto Helper
Roteamento de URLs
Criação de HTML Helpers
Validação por meio de atributos
ASP.NET MVC com Ajax
Filtros de ação
Segurança de aplicações ASP.NET MVC

Alfredo Lotar é programador e autor técnico. Desenvolve aplicações com ASP.NET, ASP.NET MVC, ASP.NET AJAX, JavaScript, jQuery, XML, XSLT, LINQ, ADO.NET Entity Framework, WCF, WPF, Silverlight, C#, Visual Basic, MySQL e SQL Server. É autor dos livros *Como Programar com ASP.NET e C#* e *Como Melhorar a Performance de Websites .NET*. Atualmente, passa a maior parte do tempo desenvolvendo aplicações, escrevendo livros ou artigos, testando códigos, e tomando chimarrão.

Fique conectado:



twitter.com/novateceditora



facebook.com/novateceditora



www.novatec.com.br

ISBN 978-85-7522-283-6



9 788575 122283 6